Technische Universität Braunschweig
Carl-Friedrich-Gauß-Fakultät
Institut für Betriebssysteme und Rechnerverbund

Collaborative Media Streaming

Von der Carl-Friedrich Gauß-Fakultät
der Technischen Universität
Carolo-Wilhelmina zu Braunschweig
zur Erlangung des Grades
einer Doktor-Ingenieurin (Dr.-Ing.)
genehmigte Dissertation

von Verena Kahmann
geboren am 8.4.1972
in Stuttgart

Referent: Prof. Dr.-Ing. L. Wolf
Korreferent: Prof. Dr.-Ing. J. Ott

Eingereicht am 15. April 2008
Mündliche Prüfung am 22. Juli 2008

Dank

Zum Gelingen dieser Arbeit haben einige Menschen wesentlich beigetragen. Bei ihnen allen möchte ich mich herzlich bedanken.

An herausragender erster Stelle sei hier Herr Prof. Dr. Lars Wolf genannt, der meine Arbeit in jeglicher Hinsicht hervorragend betreut und mit zahlreichen wertvollen Hinweisen und fachlichen Ratschlägen das Zustandekommen dieser Arbeit erst ermöglicht hat. Er hat mich als wissenschaftliche Mitarbeiterin am Institut für Telematik an der Universität Karlsruhe (TH) eingestellt und mich auch nach dem Wechsel an das Institut für Betriebssysteme und Rechnerverbund der Technischen Universität Braunschweig stets gefördert. Besonders dankbar bin ich ihm für die Freiräume, die er mir gewährt hat, um Beruf und Familie vereinbaren zu können.

Herrn Prof. Dr. Jörg Ott von der Universität Helsinki danke ich für die freundliche Übernahme des Korreferates sowie für die hilfreichen Anmerkungen zur Arbeit.

Die Zusammenarbeit mit den Kollegen, zunächst am Institut für Telematik, dann am Institut für Betriebssysteme und Rechnerverbund, war ebenfalls von großer Bedeutung für das Gelingen meiner Arbeit. An beiden Instituten herrschte ein angenehmes und offenes Arbeitsklima, zu dem viele Kollegen ihren Teil beigetragen haben. Für das Institut für Telematik seien stellvertretend Herr Dr. Elmar Dorner, der mich als Betreuer meiner studentischen Arbeiten für das Gebiet der Rechnerkommunikation begeistert hat, und die Herren Kollegen der Engesserstraße, Dr. Marc Bechler, Dr. Daniel Kraft, Dr. Frank Pählke und Dr. Bernhard Thurm, genannt. Auch während meiner Zeit in Braunschweig fand ich vielfältige fachliche und persönliche Unterstützung. Die konstruktiven Gespräche mit den Herren Jens Brandt und Oliver Wellnitz waren für die Qualität der Arbeit sehr wichtig. Herrn Dr. Bechler möchte ich auch für die Mitfahrgelegenheit nach Braunschweig danken, die immer für Spaß und Abwechslung sorgte. Dank gebührt außerdem den Administratoren des Instituts für Betriebssysteme und Rechnerverbund, welche – nicht nur während meiner Arbeit – mit großem persönlichen Engagement dafür sorgen, dass eine hochleistungsfähige Rechnerumgebung für Betrieb und Versuche vorhanden ist.

Herr Brandt und Herr Dr. Kraft waren freundlicherweise auch bereit, große Teile der Arbeit Korrektur zu lesen. Hierfür sowie für ihre hilfreichen Hinweise zum Erstellen der Arbeit mit dem Textsatzsystem LaTeX und Zeichen- und Grafikprogrammen wie *inkscape* und *gnuplot* möchte ich mich an dieser Stelle herzlich bedanken.

Zuletzt gilt mein besonderer Dank meiner Familie. Meine Eltern haben mich bereits während des Studiums stets unterstützt und ermutigt. Auch für die Hilfe bei der Kinderbetreuung bin ich sowohl ihnen als auch meinen Schwiegereltern dankbar. Frau Wacha und Frau Förster danke ich herzlich für die freundliche Aufnahme in Wolfenbüttel und die netten Einladungen zum

Essen. Schließlich möchte ich besonders das Engagement meines Mannes Julian und meiner Kinder hervorheben, die mir auch in schwierigen Zeiten immer ein großer Rückhalt waren und die während der Erstellung dieser Arbeit auf vieles verzichten mussten.

Karlsruhe, im September 2008 Verena Kahmann

Kurzfassung

Multimedia-Dienste, die mit Hilfe von Internet-Technologie erbracht werden können, sind zur Zeit ein gefragtes Thema in Forschung und Wirtschaft. Internetanbieter setzen auf Dienste wie IPTV, also die Übertragung des Fernsehprogrammes über die Internet-Protokollsuite, oder das bereits in den 1990er-Jahren entwickelte Video-on-Demand, bei dem man Filme über den Internetzugang abrufen und auf einem Fernsehgerät oder Rechner ansehen kann.

Technisch können solche Dienste unter dem Begriff *Streaming* eingeordnet werden. Ein Server sendet Mediendaten kontinuierlich an einen oder mehrere Empfänger, die die Daten sofort weiterverarbeiten und meist auch anzeigen. Über einen Rückkanal hat der Kunde auch die Möglichkeit der Einflussnahme auf die Wiedergabe: Fernsehsendungen lassen sich zeitversetzt ansehen oder die Datenübertragung kann angehalten werden.

Eine Weiterentwicklung dieser Streaming-Dienste ist die Möglichkeit, gemeinsam mit anderen denselben Film auf mehreren Geräten parallel anzusehen, egal ob sich diese Benutzer in anderen Räumen in demselben Haus befinden oder an ganz anderen Orten. Ähnliche Ansätze gibt es im Internet bereits: Beispielsweise können mehrere Zuschauer gleichzeitig einem Vortrag folgen, wenn dieser über das so genannte Multicast an ihre Geräte verteilt wird. Allerdings können sie in den meisten Architekturen keinen Einfluss auf die Übertragung nehmen - ein Anhalten der Übertragung oder gar das Überspringen unwichtigerer Teile ist nicht möglich. Des Weiteren können andere Benutzer nicht direkt über die Anwendung zu der Streaming-Sitzung eingeladen werden.

Einen *Collaborative-Streaming-Dienst* ohne die genannten Einschränkungen bietet die in dieser Arbeit entwickelte *costream*-Architektur: Sie erlaubt es, einen Film abzurufen, andere zum gemeinsamen Betrachten dieses Filmes einzuladen oder sich selbst in eine solche Benutzergruppe einzuklinken. Benutzer können die Sendung so steuern, wie sie es von einem DVD-Spieler gewohnt sind. Abhängig davon, welchen Ablauf die Benutzergruppe wünscht, wird die Steuerung für alle Teilnehmer durchgeführt oder die Gruppe aufgeteilt, sodass Benutzer den Film auch an unterschiedlichen Positionen abspielen können. Hierzu wird eine Gruppenverwaltung eingesetzt, welche den Zugriff auf Steuerungsbefehle mit Hilfe von Rollenzuweisungen regelt. Getrennt von der Gruppenverwaltung sorgt eine weitere Komponente für die Steuerung der Streaming-Sitzungen und die Synchronisation zwischen den Teilnehmern.

Eine solche Aufteilung hat den Vorteil, dass Standardprotokolle, die von der IETF für die Signalisierung von Multimedia-Diensten entwickelt wurden, eingesetzt werden können. Für die Gruppenverwaltung haben sich hierzu SIP-Konferenzsysteme als optimal erwiesen, die Sitzungssteuerung lässt sich mit der Erweiterung eines RTSP-Zwischensystems, wie man es beispielsweise auch für das Caching von Mediendaten benutzt, bewerkstelligen.

Die Evaluierung dieser Architektur zeigt schließlich, dass ein solcher Dienst nicht nur mit sehr geringen Wartezeiten erbracht werden kann, sondern eine durchaus akzeptable Synchronisation der Datenströme auf die verschiedenen Ausgabegeräte der Benutzer erreicht werden kann. Zudem ist der nötige Zusatzaufwand verglichen mit üblichen Konferenz- oder Streaming-Systemen sehr gering.

Abstract

At the time being, multimedia services using IP technology are a hot topic for network and service providers. Examples are IPTV, which stands for television broadcast over a (mostly closed) network infrastructure by means of the IP suite, or video on-demand, which allows for watching selected movies via Internet on TV devices or computers in the home.

Technically, these services can be classified under the notion of *streaming*. A server sends media data in a continuous fashion to one or several clients, which consume data portions as soon as they arrive, mostly displaying them also. By using a feedback channel customers may influence the play-back, since they may watch programs time-shifted or pause the program.

An enhancement of such streaming services is to watch those movies together with a group of people on several devices in parallel, independent from the location of the other group members. Similar approaches have been developed using IP multicast, for example for distributing lectures or conference talks to a group of listeners. However, users cannot control the presentation: pausing or skipping of more unimportant parts is impossible. Moreover, the streaming presentation is announced by means outside the application instead of adding others to the session directly within the application.

The *costream* architecture developed in this works offers a *collaborative streaming service* without these limitations: People may retrieve movies, join others watching a movie or invite others to such a *collaborative streaming session*. Participants of a collaborative streaming session can control the movie presentation like they do on a DVD player. Dependent on the desired course of the session the control operation is executed for all users, or the group is split into subgroups to let watchers follow their own time-lines. For this, a group management controls access to session control operations by means of user roles. Separate from the group management, the so-called *association service* provides for streaming session control and synchronization among participants.

This separation of duties is advantageous in the sense that standard components can be used: For group management, SIP conferencing servers are suitable, whereas session control can best be handled using RTSP proxies as already used for caching of media data.

Eventually, the evaluation of this architecture shows that such a service offers both low latency for clients and an acceptable synchronization of media streams to different client devices. Moreover, the communication overhead compared to usual conferencing or streaming systems is very low.

Contents

List of Figures

List of Tables

1. Introduction

Streaming of media presentations has attracted much attention in the last few years. Video-on-demand services have already been invented in the nineties of the last century, however, without big commercial success. Despite this, new standards like MPEG-4 have been invented to enable a more flexible data representation together with efficient coding formats suitable for streaming of media data. Recently, television broadcast using Internet protocols (IPTV), or the provision of television, telephone, and Internet access from one source (so-called Triple Play) have gained much interest, opening up new perspectives for media-on-demand. Particularly the possibilities for interactive control such as pausing a movie are appreciated as an advantage over plain television without feedback channel.

Even for mobile systems, streaming services have been standardized in mobile phone systems of the third generation. Efforts in the provision of broadband access networks and in guaranteeing quality of services in different network environments inspire the vision that accessing media content anytime, anywhere is not far beyond reach anymore.

One type of a media-on-demand service that is useful in a range of networking environments is a *collaborative streaming service*, which allows its users to join and watch streamed media presentations together. Several scenarios benefit from such a group streaming service, in the consumer entertainment sector as well as in professional environments like e-learning. Unlike available solutions, the architecture developed in this work is applicable to a range of different scenarios, independent from the particular network settings.

1.1 Motivation

Media-on-demand streaming services are of interest for different scenarios. In the networked home, where IPTV or Triple Play as mentioned above exists, several different consumer electronics devices can be used to watch streaming presentations. Inhabitants may choose to watch streaming presentations on these devices in parallel. In educational environments, groups of learners watch lectures recorded onto streaming servers. The need for cost-savings in business requires new means for advanced training: People working at different company locations take part in training sessions. Learning material for such scenarios can be retrieved from a media server.

In all of these scenarios, a group of users, each one controlling a number of devices, joins to watch particular content in the form of a streaming presentation. During the course of the presentation, users want to control it according to their needs. Like with a remote control of a video recorder or DVD player, the streaming presentation can be paused or played with different speed, the audio track can be switched off or to a different language, and so on.

Thus, users have the common interest of watching the presentation. In contrast to unrelated groups of mass media distribution, they share the common interest of discussing the content of the presentation or gaining knowledge from it. However, human beings differ in their perception of content: Different people consider different scenes of a movie as important. In learning environments, people have different learning speed. From this it follows that a collaborative streaming architecture has to satisfy both requirements for a shared manageable group relationship and for individuality.

1.2 The Notion of Collaborative Media Streaming

Until today, no exact definition of *collaborative media streaming* exists. *Media streaming* is defined as transferring packetized media data from a source to a client, which receives, reassembles and displays data portions as they arrive from the server. Throughout this whole document the notion of streaming is used in the sense of *real-time* streaming: The transfer and the display of media data have a strict time relation, which also means that media streams are transferred at about the same speed as the stream is played out. This is useful for systems with restricted buffer capacities.

Streamed media *presentations* either are delivered from live sources or from media files stored on a server. Individual position control is in our case meaningful for stored presentations only. Presentations have a *description* giving information about the parameters of the session itself and the contained media. This can be technical information as well as semantic meta data. The latter, however, lacks a unique, widely used standard.

An important notion correlated with real-time streaming is the *session* which is used as the abstraction of the relation of server to client. Each session has a certain identification and a session *state*, which describes changeable attributes of the presentation as delivered by the server. Such attributes, like the viewing position or the track selection, are influenced by control operations.

In this thesis, a *collaborative streaming session* is defined as the set of the streaming sessions of the same presentation to a group of clients at the same time. Each streaming session may have a different state. Operations on this state must follow a common group policy. Consequently, the set of participants together with the collaborative streaming session and the group policy forms the *collaborative streaming group*.

1.3 Focus and Approach of this Thesis

Collaborative Media Streaming has been addressed only in a very limited fashion until today. In most architectures, content distribution to a number of participants is the main concern. On the one hand, collaborative streaming in the Internet is often considered equivalent to session sharing by multicast streaming. On the other hand, existing multimedia middleware architectures – which provide for programming abstractions for recording, processing and displaying of multimedia data on a range of devices – concentrate on establishment and management of graphs for a shared data path. Both points of view limit the applicability of session control operations for a group of participants.

In this thesis, the collaborative group and its participants receive priority. We examine how a collaborative streaming group is coupled in different scenarios. Practically spoken, a group will

split, merge, or change session state as a result of different control operations. Other operations do not affect the group, but are executed individually. Another important issue to resolve is who is allowed to execute control operations that affect the group.

To approach these issues, the state of a streaming session which can be changed at all is examined. At the time being, this state consists of the set of media tracks and the play-time position. Often, the selection of a certain track set is important for the individual quality perception of a user, whereas other members are not affected by this selection. Hence, the notion of *shared session state* is introduced to make a differentiation between the part of session state which is controlled by the whole group and the other part which is controlled by individuals.

As interests inside a group concerning shared session control may differ, it is reasonable to let a group split into smaller entities if desired. An *association* consists of the participants of a collaborative streaming session which share the same state inside this session, e. g. accessing the presentation at the same point in time-line.

These associations inside a group are formed by control operations on the shared state. For many scenarios, a *policy* to regulate such control operations is important, e. g. to avoid splitting into too many associations. In this thesis, a concept for such a control policy is presented. Participants are assigned *roles* with a specific permission set, which consists of session control operations. Since neither the policy of always splitting a group nor the policy of always denying a control operation is sufficient, a differentiation between different *reactions* to a control operation is made: The shared session state can be changed for the whole group, or a user can open an own association so that the group is split.

Finally, an architecture is required that allows to implement the concepts mentioned above and is deployable in a number of different scenarios. The IETF application-layer multimedia signaling protocols SIP and RTSP are chosen for this. Particularly the separation of control and data path facilitates design. Moreover, a number of standards already exist that allow to map the required collaborative control operations to operations for conferencing using SIP methods on the one hand, and for retrieval of streaming media from streaming servers using RTSP on the other hand.

1.4 Contribution of this Thesis

The goal of this work is to present a comprehensive architecture for the collaboration of several clients watching a streaming presentation. This architecture is applicable in a number of network settings, particularly Internet environments.

Since groups may be formed in a dynamic fashion, primitives that manage such collaborative streaming groups have to be implemented. Clients are added to or removed from collaborative streaming sessions on their own initiative or by other participants. Collaborative streaming sessions can thus be pushed to other clients or pulled from other clients. The architecture presented in this thesis provides for session transfer primitives for that purpose.

By using IETF standard protocols, the architecture is deployable without changing any application on a remote media server. It is not required to install a certain middleware architecture everywhere. Instead, it is possible to enhance standard components such as RTSP proxies and SIP conferencing servers by the modules for session sharing and group management, respectively.

Particularly, we have contributed the following components:

Session sharing uses a so-called association service which is implemented on top of an RTSP proxy. This service manages streaming sessions on behalf of collaborative group participants, which are grouped into associations. Streaming session management also means that the association members receive media streams synchronized with those of other participants. This is achieved using different synchronization modes.

Session control is implemented by the association service and requests policy decisions from the group management if an association contains more than one member.

Session transfer primitives allow to manage collaborative streaming groups, i.e. instantiate them, add or remove participants, and so on. These primitives are visible to clients as well as they are implemented within the group management application.

Group management implements the session transfer primitives on server side and also manages the control policy for each group. It delivers a decision about a control operation, which can be forbidden, used to change state of the whole association, or used to split the association.

Though being interesting for the provision of a collaborative streaming service, some aspects are out of the scope of this thesis.

Quality of Service and Content Distribution Techniques Even though a certain service quality is an important aspect for the success of a service, we will not elaborate on approaches to provide for QoS, because the existence of these mechanisms is transparent to the signaling for collaborative streaming.

Research has been done for distribution of content to a mass of consumers in an efficient fashion. However, those media data transport mechanisms are also transparent to the collaborative streaming service, which is supported by the separation of control and data path in the architecture presented in this thesis.

Security and Content Protection Security is an important issue for each work in distributed communication. However, several IETF protocols like SIP and RTSP already expose mechanisms to security, besides that network-layer security mechanisms like IPSec exist. Content providers fear that content is copied among users. Though preventing users from saving media data on their systems and redistributing it to others is beyond the scope of this work, the architecture generally allows providers to install security mechanisms on the intermediate system.

Meta Data for Presentations and Devices Though meta data are important for the description of streaming presentations as well as device capabilities, no comprehensive standard that is widely deployed exists today. The most important technical parameters of a streaming presentation are described by the presentation description, but content identifiers utilizable for both user- and machine-friendly content searches are missing. Signaling device capabilities is another aspect that is completely missing in streaming systems of today, though particularly the transport efficiency of collaborative services can benefit from this, because heterogeneous devices can be served according to their needs.

1.5 Outline

Subsequent to this introduction, selected scenarios in which collaborative streaming occurs are described in chapter 2: home networks for entertainment, learning scenarios and mobile spontaneous meetings. The differences of their networking environments together with already

available standards and services are examined Moreover, differences and common denominators of participants' interests are presented. This leads to a common set of use cases and requirements for a collaborative streaming architecture.

Related work is presented in chapter 3. Though only few approaches for collaborative streaming architectures exist at all, a number of related concepts is presented. Middleware architectures in closed network environments are used for session sharing purposes. Partially, even transfer of a particular group state to other devices is supported, which is relevant in conferencing environments. Other collaborative work approaches provide for group management concepts, which often include the notion of different views on a common group interest – similar to what is required in collaborative streaming.

These concepts are pursued further in chapter 4, where the basic concepts of collaborative streaming are defined. The management functionality for group state is presented, which comprises access control as well as the enforcement of a consistent state for control operations. Another relevant issue is how such control operations actually influence a streaming session. Finally, the practical effects of session sharing, particularly inter-client synchronization, are discussed.

Following from these concepts, the notion of a collaborative streaming service is introduced in chapter 5, highlighted from the user and from a system perspective on the service. The user requires to control the group membership as well as the streaming session, which means that transactions for initiation of the group as well as for update of the session have to be implemented by the system. This is done by a composition of separate services for group and session management.

Leading to a concrete system architecture, the relevant IETF multimedia signaling protocols RTSP and SIP are described in chapter 6 in detail. Additionally, the media data transport protocol RTP and approaches for configuration of multimedia services in the Internet are delineated.

The collaborative streaming architecture *costream* as the main achievement of this thesis is presented in chapter 7. The separation of the services by means of intermediate components is shown. Afterwards, the general message flows for initiation and update transactions using SIP and RTSP are described in detail, before the modular design of the service components is presented.

In chapter 8, we describe the most important implementation aspects of our proof-of-concept prototype. Third-party components and libraries that have been used are described, followed by the definitions of the most important interfaces and classes of the individual components.

Finally, we provide for qualitative and quantitative evaluations in chapter 9 before we conclude this thesis with a summary and an outlook on future topics of collaborative streaming.

2. Scenarios and Base Technologies

Collaborative Media Streaming is useful in a number of different scenarios. Different ways how people in home, learning and mobile environments watch a media presentation together and inter-operate during the course of the streaming sessions are identified in the following sections. Examples show in detail which collaborative streaming functionality is required in a certain scenario. Besides, technical preconditions that characterize the particular environment are presented, as well as existing services in the environment that are related to streaming or collaborative work.

Eventually, a general scenario of collaborative media streaming is composed in section 2.4. In this section, common functionality for collaborative session initiation and management is examined. Moreover, requirements and conditions which are relevant for this thesis are described in section 2.5.

2.1 Home Environments

In home environments, media streaming is obviously used for watching movies. Instead of buying or renting a DVD, the movie may be streamed on demand from a server on the Internet. Currently, more and more digital television content is offered by service providers using computer networks with the Internet Protocol, the so-called *IPTV*, which is currently under consideration for ITU-T standardization efforts also [73]. Dependent on the service that provider companies offer, the users may select specific movies (called *video-on-demand*) or record the TV program to a Personal Video Recorder (PVR) [112], which can also be used as a home server to stream media data to TV devices in the home environment. Service providers may offer such PVR in combination with proprietary set-top boxes to allow access via the *closed network* of this provider only. This approach is followed by cable providers mostly (e. g. Kabel Deutschland [56]). In contrast to this, telephone providers (like T-Com [28] or Arcor [25] in Germany) often offer video-on-demand services based on open standards, which enables the subscriber to download or stream a movie from the provider's Internet portal in a certain time-frame. The integration of television content, telephone services and Internet access (both using a common network architecture and forming commercial offers for customers) is subsumed under the current headword of *Triple Play*. Collaborative aspects in home environments come into play when people watch a movie on several devices in parallel and copy or move their movie session from or to different devices in the networked home. Existing video-on-demand services allow to watch a movie on several devices within a certain time-frame, but do not support functionality for session transfer.

We assume in the following that a number of networked media devices exist in different rooms of a private home. At this point, we do not make any assumptions about the nature of the streaming service, and we leave out licensing issues.

2.1.1 Collaborative Streaming Characteristics

Consider, for example, a family watching a movie in their living room. The teenage daughter stays in her own room with a TV set, and she just likes to find out which movie the others are watching, so she joins the film session already in progress. A friend of the family who owns a PDA with wireless LAN interface visits the family and is invited to attach the PDA to the home network and join the presentation. Since the mother tongue of the friend is French, he chooses to receive the audio stream of the movie in French language via the earphones of the PDA.

We extend the notion of collaboration not only to be applicable among persons but also among devices. Hence, we can also define the case of moving a session to be a collaborative action. Imagine a person that is doing housework and thus has to go from one room to another, but still likes to watch the TV entertainment program, moving it from one display to another. Similarly, family members that have started watching a movie on a mobile device may choose to move their session to a higher quality display when they come home.

Thus, several collaborative interactions can be identified in home environments. People can join or be invited to join a streaming session, as shown in figure 2.1. Streaming sessions can be moved to other devices or retrieved from different devices. This form of interaction is called *session transfer*, which will be described in further detail in section 2.4.1. Home users want to execute session transfer actions as intuitively as possible, thus the architecture must support users in finding devices and available sessions.

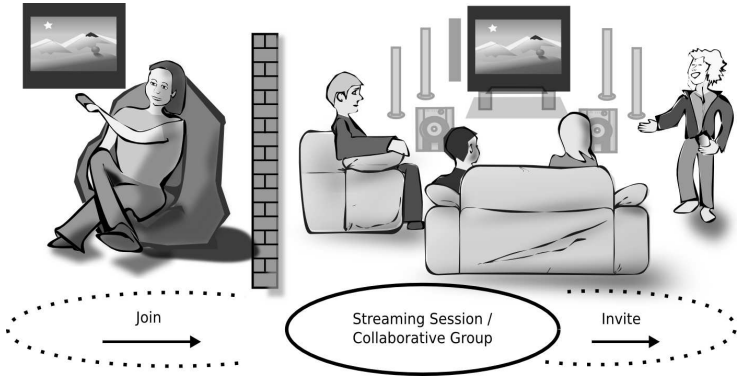


Figure 2.1: Session Transfer in Home Scenario

People with separate devices can choose their own parameters, like the foreign language in our example. It is also possible to configure different quality settings like video formats, which will likely have to be done if a heterogeneous set of devices has to be served. However, in a home environment such settings are mostly pre-configured and will not be changed very often.

Video-on-demand streaming services of today rarely offer different quality settings, but switch to downloads of movies coded with a limited bandwidth (T-Online currently offers 800 kbit/s, e. g. [28]). For users, an intuitive information about available parameter settings (like high or low quality) can be advantageous, but the achieved quality is unsuitable for a number of devices.

For recorded streaming sessions, a session may be controlled with respect to the play-time. Such *session control* works like a video recorder control, with functions to pause the movie or to jump to a later point in the time-line. However, the question comes up how session control will work if a session runs on several devices, like in the example presented at the beginning of the section. In figure 2.2, an exemplary scenario is shown: For the daughter, the movie is not too interesting, thus she just jumps to a later point in the movie time-line to see the end of the film. The rest of the family is not affected by her behavior and can watch the movie without interruption. Later, the mother wants to get dinner from the kitchen, so she asks to pause the movie. Both sessions on the main TV screen and on the PDA are paused using the remote control of the main TV.

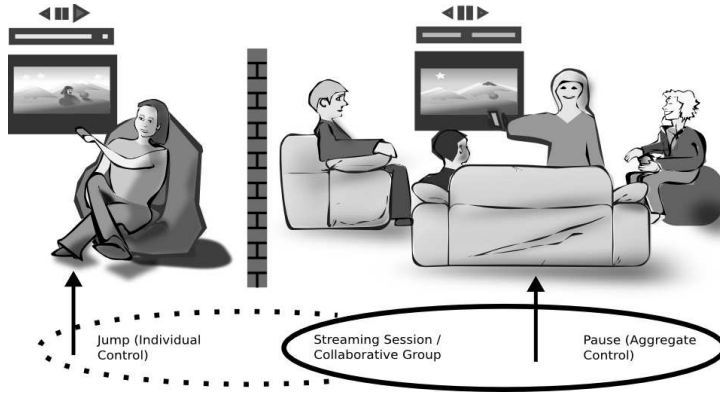


Figure 2.2: Session Control in Home Scenario

Thus, in a collaborative home scenario, the different devices may be controlled in an *independent* fashion so that the daughter can individually change the play-time position without interrupting the session on the main display. In other cases, *aggregate* control is useful, in our example when the sessions of main TV screen and PDA are paused together. An important task of a collaborative streaming architecture is to define when aggregate or independent session control should be used. If several devices in one room are used, aggregate control is the normal mode of operation, but for different rooms both forms of control may appear, depending on the users' demands.

Another aspect of collaborative streaming that is important in the home environment is the transfer of the common session state. Session state means all parameters that can be influenced by a client during a streaming sessions. Examples for such parameters are the *viewing position* or the *selection of media streams*. Members of a collaborative streaming session normally have a common view on these parameters, and those members who join a collaborative session at a later point in time should be equipped with the state that is relevant for the group at the time of joining. Particularly for the viewing position as a dynamic parameter this is an interesting task, because sessions running on different devices must synchronized. The transfer and maintenance

of the common session state is called *session sharing*. Session sharing is also required at the time a collaborative session control command is executed, for example at the time a group member wants to synchronize to another group member that watches the movie at a different play-time position.

Finally, some user-related characteristics of collaborative streaming applications for the networked home are to be mentioned here: In home environments the number of users is relatively low, ranging from one to about 5-7 users. Social interaction among people is an important aspect, thus some family members may rather talk about who should take session control than configuring the system. Another point is that expertise differs among users, and expertise of a network or technical administrator cannot be taken for granted. To summarize, users prefer technical systems that support their needs in an uncomplicated, unobtrusive fashion.

2.1.2 Technical Conditions

Networked home environments consist of a variety of different consumer electronics and computer devices with network interfaces. Even domestic appliances are sometimes equipped with network connectivity. Vendors interconnect these devices to sell home automation platforms to form smart or intelligent homes. However, such specialized networks are not considered here. Instead, this work concentrates on home networks as an extension of the global Internet.

Other specialized platforms handle the delivery of media data, for example the *Multimedia Home Platform* (MHP) [102]. Many of these platforms have implemented TV and broadcast functionality until today. Thus, they can form a middleware for delivery of TV content using computer networks (IPTV). The high number of subscribers for Internet access and the competition between service providers forced the idea of integrating access to the global Internet with these platforms. Applied standards are driven by manufacturer and vendor consortiums, and interoperability among standards is not the main goal. An unique standard for the integration of multimedia middleware with Internet access has not come to widespread use until today. However, since there is a growing interest in integrated services like Triple Play, this may change in future.

2.1.2.1 Devices

Traditionally, the set of devices in a home can be classified into separate groups: first domestic devices (which are not considered further in this work), second computer devices, and third consumer electronic devices like TV sets. However, the increasing interconnection of devices will lead to more inter-operation or even to a merging of the classes in future.

Devices in a multimedia home network comprise capture devices like cameras and microphones, output devices like video and text displays and speakers, and processing devices like computers. For interconnection means, switches or wireless access points are used. If a connection to the global Internet is used, a cable modem or a DSL router must be available. Such routing devices may contain intermediate devices like switches or access points.

The device capabilities are generally heterogeneous, for example display sizes may differ from wide-screen displays over computer monitors down to small mobile device displays. Media client applications also span a broad range from software to hardware players. An example for the latter are the so-called “streaming clients” which are used to play media files from repositories on a TV or HiFi device. For media streaming, play-out delays are an important aspect. An important

part of the end-to-end delay is caused by buffering, which is needed for decoding and display on output devices. Sound cards use buffers composed of multiple parts, also called fragments, to be capable of concurrent input and output. The size of these fragments is dependent on the card, an example value is 1024 bytes. The number of fragments should be low, the best value considering latency would be 2. Operating systems normally offer to tune the values of fragment numbers and size. The resulting latency of usual sound cards with e. g. the DirectSound driver is higher than 20 ms. More sophisticated audio device drivers like ASIO feature delays of 2 - 10 ms depending on the used audio card. However, these are used in professional environments rather than in the home.

2.1.2.2 Networking Aspects

To analyze networking aspects of home environments, two different connection types can be distinguished, first the in-home network itself, and second the connection to the global Internet. Home devices are interconnected using a number of different networking standards, of which the following are best known:

IEEE 1394, also known as *Firewire*. This standard allows to plug a number of different, mostly peripheral devices to a computer. It also allows the formation of a complete home network. The available bandwidth is at most 400 Mbit/s for IEEE 1394a [61] and 800 Mbit/s for IEEE 1394b [62] compatible devices. The distance is restricted to 4.5 m per device and 72 m overall. The underlying protocol is not full-duplex capable, thus the latency for interactive applications can get high, especially if many devices use the network. Due to the channel arbitration procedure, which is done every 125 us, an isochronous service without jitter can be offered.

Phoneline Networking uses the available phone lines in the house. With the new HomePNA standard, which is also approved by the ITU [70], data rates up to 240 Mbit/s (today available are 128 Mbit/s) can be offered. A drawback is that phone jacks are normally not available in all rooms.

Powerline Networking comprises technologies to modulate data to the low frequency oscillations of power lines. The advantage for in-house use is that no additional network cables are required, and power sockets, in contrast to phone jacks, are available in every room. The HomePlug standard [57] offers data rates up to 14 Mbit/s. Powerline communication has not found widespread acceptance until today, because of the more expensive devices compared to other technologies. Another drawback is the interference with radio services and the possibility of wiretapping because of the transmission over unshielded cables.

Ethernet as an implementation of the IEEE 802.3 standard [63], is a widespread computer network standard. The available bandwidth ranges from 10 Mbit/s to 1 Gbit/s, however cards running at 100 Mbit/s are presently used most. Ethernet offers no guarantees for bandwidth or latency if several devices share a segment due to its CSMA/CD medium access control (MAC) procedure. However, affordable switches are available, which separate segments so that each device can use an exclusive segment. Up to now, only few consumer devices except some cameras are equipped with Ethernet adapters.

IEEE 802.11 WLAN has been designed as a wireless extension to local area networks following the IEEE 802.x standards [60]. At present, mainly computers, but also a growing number of devices like printers, cameras or phones are equipped with the cheap WLAN cards. The bandwidth ranges from 1 Mbit/s to 54 Mbit/s [64], which is also dependent on wireless

radio connectivity. Since the MAC procedure is similar to Ethernet, however with a backoff timer at the sender to avoid collisions (denoted by the term CSMA/CA), and all devices sending on the same frequency share the medium, the usable bandwidth is (in a relative way) lower and delays are higher than in a wired Ethernet. Extensions with higher bit rates and a modified MAC layer for quality provision are ready for marketing.

Other standards like USB, Bluetooth or Infrared to connect peripherals like control devices may be used also, but are less suitable to build whole networks.

For the connection to the global Internet, telephone lines have been used traditionally. Whereas ISDN offers only 2 lines of 64 kbit/s each for the home user, the development of mechanisms like DSL (Digital Subscriber Line) [69] has led to download bandwidths of 1 - 16 Mbit/s for typical home use, and up to 25 Mbit/s for test use. DSL is available either asymmetric, with a lower upload bandwidth, or symmetric with equal download and upload speed. DSL has found world-wide acceptance, with 11 million subscribers in Germany and 9 million subscribers in the U.S. in 2003 [9], but requires extension of switching centers and availability of copper wires at the local loop to the subscriber. However, home users can install the required equipment without help of a technician.

DSL is not available at locations where fiber has been installed as an access network (for example in the Eastern part of Germany, where FTTC (Fiber to the Curb) replaced copper wires), because the technology is based on using higher frequency bands of the copper wire. Since solutions like VDSL (Very High Data Rate DSL, which may use a hybrid fiber/copper network) or the installation of outdoor access multiplexers are quite expensive for the telephone companies, customers use Internet access via TV cable as an alternative. Especially the widespread availability of the cable network has made this access method interesting for providers. While technical premises allow similar download rates to DSL, restrictions to the cable market as well as the insufficient cable network without a back channel prevented the widespread acceptance in Germany in the past. Thus, only about 200,000 subscribers used cable modem there, compared to about 15.0 million subscribers in the U.S. in 2003. Globally, cable modem was with 33 million subscribers the second broadband Internet access technology behind DSL (58 million). ISDN, in comparison, had 127 million users in 2003. The subscriber numbers for 2004 were growing to 81 million DSL vs. 43 million cable modem vs. 133 million ISDN subscribers [9]. Users need a cable modem which can be connected to a computer via USB, Ethernet or WLAN. The modem is mostly sold by the provider, because different methods to use available cable frequencies existed historically. A possible disadvantage compared to DSL is the fact that the network is shared with a number of other subscribers. On the one hand, this restricts the usable bandwidth, on the other hand, encryption mechanisms must be used to protect privacy of users.

We show a schematic overview of a home network scenario in figure 2.3. A variety of in-home devices is connected via a DSL Customer Premise Equipment (CPE) or a Set-Top Box (STB) to a Cable or DSL Provider Gateway. This provider either uses own media servers or offers services from third-party media service providers, with an arbitrary, but high-bandwidth, Internet connection.

Media streaming from the Internet or digital TV providers to the home network is thus possible, however there is still a gap between available bandwidth in the home network and the access network. Hence, efficient coding techniques like MPEG-4 are important for the provision of streaming to the home. The collaborative streaming architecture also must account for

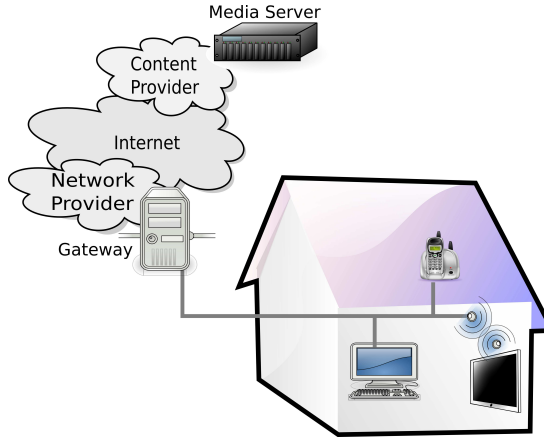


Figure 2.3: Schema of Home Network

the restricted bandwidth. In home scenarios, synchronous play-out on different devices is a requirement. Thus, networking delays should not differ significantly. Inside local area networks, delays are normally lower than 10 ms. For Internet access, latencies vary due to the used technology. Dial-up modems experience delays of more than 100 ms, whereas broadband access like DSL normally feature tens of milliseconds of delay [75]. Since the access connection is supposed to be the same in one home network, and in-home networks are supposed to use LAN technologies, devices do not experience significantly different delays in their connections to the Internet. We therefore expect that different device and application characteristics have a higher influence on synchronous play-out than the present networking differences.

2.1.2.3 Multimedia Middleware and Services

In home networks, multimedia middleware tries to integrate a number of different audiovisual devices. The goal of such an integration is to offer enhanced services to the home user and to limit programming effort for devices. As an effort towards interoperability, which is another goal for consumer electronics vendors, the *Home Audio Video Interoperability (HAVi)* standard [54] has been specified by a group of product producers to provide for resource sharing and integrated control of several consumer devices. It uses IEEE 1394 as underlying networking technology and provides an open API where portable distributed applications can build upon.

The *Multimedia Home Platform (MHP)* [102] has been defined as a middleware to deliver digital interactive TV and, as an enhancement, Internet access to home users. MHP has been initiated by the Digital Video Broadcast (DVB) project. For the home user, it comprises the MHP terminal, i.e. a set-top box (STB) on which applications run. There are three profiles specifying STB capabilities: The Enhanced Broadcast Profile provides for digital TV, the Interactivity Profile adds interactivity features to this by using a return channel, whereas the Internet Access Profile of the MHP 1.1 specification allows for advanced Internet access, possibly using cable networks as mentioned before.

MHP offers a so-called *Personal Video Recorder (PVR)*, which is used to record content, mainly from television programs. Compared with usual Video Cassette Recorders (VCR), it provides for advanced functionality like cutting advertisements and pausing live TV.

The so-called *Triple-Play* service combines broadband Internet access, entertainment services like Video on Demand (VoD) or IPTV, and telephony services over IP. Telephony providers are interested in binding customers by offering services from one source with certain cost savings. T-Online, for example, offers different triple-play packages using ADSL+ or VDSL variants, ranging from VoD services to the reception of about 50 free-TV programs. Cable providers also have the infrastructure to offer this service, if back channels can be used. IPTV or VoD services for the home user can be implemented as so-called *PushVoD* service: Broadcasters and providers use multicast or carousel delivery mechanisms to distribute desired movies to a potentially large number of subscribers that record the content on their PVRs. Advances for multimedia delivery over mobile phones have created the notion of *Quadruple Play* which means Triple Play with the integration of mobile and wireless services. Such Quadruple Play services could be used to provide for handover of video play-out from the mobile phone to the home TV display.

At the present time, enhanced services for in-home collaboration among devices are relatively rare. Simple control functions are provided by device vendors and can be accessed either by the above-mentioned middleware or by protocols like SOAP [168]. However, only few projects [6] implement parts of session transfer and control functionality as required for collaborative streaming.

2.1.3 Requirements for Collaborative Streaming in Home Environments

Collaborative streaming functions are session transfer and session control (confer section 2.1.1). A more detailed description of these functions will follow in section 2.4, because both are required in every scenario. An important question is who should be allowed to execute these actions. Since not all users in home networks are experts, and social interaction takes an important part, such rules for controlling access to the collaborative streaming service should not be too difficult. Yet it is reasonable to provide some system support, for example to prevent children from starting sessions or joining movies intended for adults only. Similarly, for session control such support can also be reasonable in cases where face-to-face communication is not possible. The exact definition of an access control model suitable for several scenarios will be presented in section 4.2.1. We state here that restrictions for starting, joining and controlling sessions should exist. In case of home networks, these restrictions may depend on the user that executes the action, on the type of content, and on the type of device. Consider, for example, the case of the foreign-language visitor, who may join the session on invitation only and – due to his audio-only device – always follows the control actions of the rest of the family.

Another setting that must be defined is how to execute session control: in an aggregate or independent fashion. In a home network, executing any position control will normally cause all members of a collaborative session to follow this control request, at least if all people are in one room. People in different rooms may use their own play-time position and thus create new subgroups on demand. Otherwise, users would be annoyed of anyone influencing their presentation. Persons in different rooms should have the possibility to re-synchronize with the group.

The selection among different tracks of one presentation is driven by device and user requirements, which may be different for each single device or user. An example for this is the foreign-language

visitor mentioned before who is streaming the audio track in his mother tongue while watching the video track on the home display. Translation engines are not available until today, thus the different track must be sent. However, it is important to note that the video track must not be sent to the PDA if only the audio is required. Thus a *flexible track setup* is needed.

As already mentioned, families interact by talking to each other rather than by using technical means. Hence, voice or text chat channels may be reasonable for family members in different rooms. However, the possibility to install such channels depends on available networking resources and on the capabilities of their devices. We assume that such channels are installed if enough bandwidth is available in the home network and that these channels can be switched on and off on demand.

Configuration of devices and services should be as easy as possible in a home network. Hence, the user interface for home users should provide for user-friendliness above all. It is annoying for users to type in addresses on possibly input-restricted devices. Instead, *automatic location* of services and other user devices should be possible. Concerning the management of the rules mentioned above, client device interfaces should not show actions which may not be executed. The rules should be adaptable, but only by the administrator. Since the administrator could change at every session, it is convenient to keep configurations in different files and upload them where necessary.

In home environments, *synchronization* needs to be quite strict because several streaming devices may be visible to the user. Even more, lip synchronization between different devices must be possible, because one device may be used as an audio-only device while streaming the video track to a different device. This means that audio and video streams of different devices should not drift more than 80 ms to be in the range of user desirable values [152]. This value range also depends on the type of movie that is shown, for example synchronization skews in movie scenes showing landscape sceneries with some music as audio stream are not as evident as in scenes where talking people are shown in close-up.

Efficient usage of resources in home networks is important, particularly for the possibly expensive connection to the global Internet. Due to the different capabilities of devices available in a home environment, clients will likely stream different sets of media tracks onto their devices. For example, different display sizes are best served by media tracks conditioned to the resolution of the respective display. In such a case, an efficient data transport would deliver only the best-quality stream to the home network and use *layered coding* [99] or *transcoding* [162] techniques in an intermediate system in the home to recalculate the media data for the smaller devices.

2.2 Learning Environments

Computer-aided learning is an interesting part of school, university and professional education. Since use of multimedia material is considered to be important for efficient learning, several platforms and standards have emerged to compose textual and audiovisual blocks to learning courses (confer also section 2.2.2). On the other hand, learning in groups of students is done mainly at schools and universities, where the discussion of themes contributes to learning success. However, also vocational training can benefit from team discussions about educational presentations. Thus, a learning scenario making use of collaborative streaming technique is discussed in this work. This scenario enables groups of students to consume and discuss recorded audiovisual material.

Collaborative streaming can be used for learning sessions at certain points in time, i.e. when a group of users meets to process and discuss streamed content. Thus, the location of the learners can be decoupled, but the time of the collaborative streaming session will be fixed. This is useful for companies which may provide distributed company sites with training programs, because participants do not have to travel to a certain location. In this context, the individual expertise of learners can also be considered, because learners with greater knowledge can skip chapters whereas beginners may require additional information. Moreover, learning teams can be formed to elaborate and discuss parts of the presentation. Training programs can easily be tailored to the needs of employees and thus save costs and resources for companies.

In professional environments, collaborative streaming can be applied in the context of on-line instructions. For example, a technician at a remote site may ask for support from a company site which is available as a streaming presentation. If questions occur, the technician may ask an expert to join the session and to share his / her expertise.

A special collaborative learning application is the distributed presentation where conference participants present their material to a learning group. This is a special case which differs from our general scenario, because participants are not only consumers, but also media producers. We use the simplification that such users offer their presentations using a streaming server at their own site, which generalizes this case to a usual live streaming case.

Collaborative streaming is an enhancement of training programs or interactive live lectures in the form that content may be discussed among learners using chat or audiovisual sessions and that an enhanced group management is used, which allows the partitioning of groups to elaborate different topics or to adapt to individual learner speed.

2.2.1 Collaborative Streaming Characteristics

We assume that collaborative learning is mostly started with a fixed schedule, on initiative of a single (teaching) person. Students join the session, for example following a hyperlink. Such a scenario is shown in figure 2.4. In this case, the group of participants can be set up in advance. However, for training scenarios starting a session alone is reasonable also. In both scenarios, late joining members should be allowed to add themselves to the learning sessions. A supervisor or the person who has started the training session should have the possibility to invite others to the session.

Normally, groups are not open to the public, i.e. only users which have successfully registered to the system can access (collaborative) streaming presentations. If no difference between specific presentations is made, no access control for single presentations must be provided. Otherwise, any request to join must be checked against the set of users registered for a specific presentation. Session transfer is thus handled more restrictive than in private home environments.

Similar thoughts hold for session control. In learning scenarios, it can be assumed that one person takes the role of a supervisor and controls all actions of the group. This means that jumping to different points in time-line or setting up semantically different tracks should only be done on demand of this supervisor. This case is shown in figure 2.5, where the supervisor forms two subgroups by directing students in different classrooms to different units of the learning material. Pausing, for example to ask questions, should be allowed at any time. Obviously, such rules can also be changed, for example if users over-use the pausing possibilities. Since different course types can use collaborative streaming, it is possible that either a group follows a pause request,

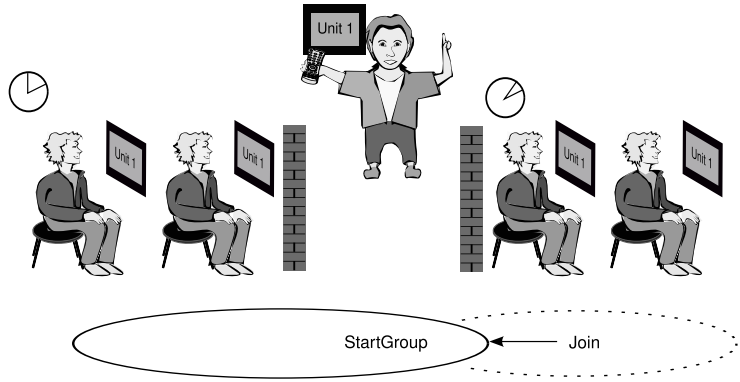


Figure 2.4: E-learning Scenario

e. g. in lectures, or the requester creates an own subgroup by changing the play-time position, in cases where certain material should be learned by oneself. However, these actions are again controlled by a supervisor or trainer.

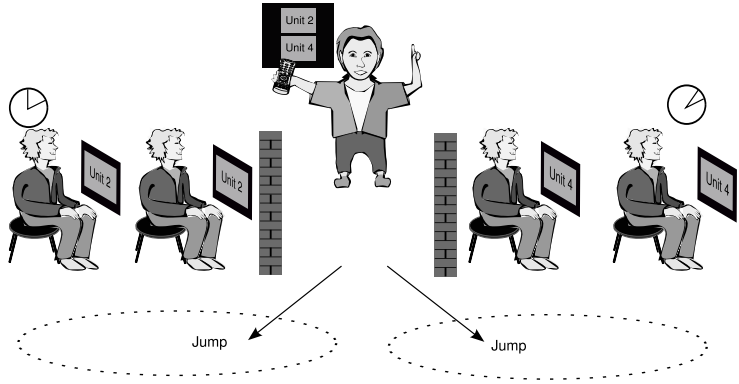


Figure 2.5: Session Control by a Supervisor

For technical conditions of learning, the assumption can be made that distributed learning takes place in distributed learning centers, e. g. schools or university departments, or at the local work place at a company site, as shown in figure 2.6. In a local learning center as depicted by the rounded boxes, several learners may share a classroom. Several work places at a company site can be interconnected using a local area network. A gateway will be used to connect the learning center to the Internet or to other learning centers. Mobile learners may also be connected to a learning center, like in the figure on the left side, using this gateway.

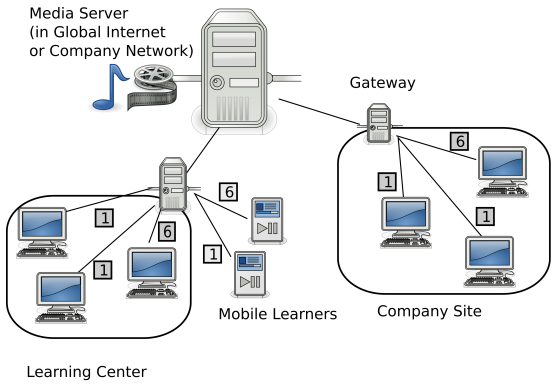


Figure 2.6: Overview of Learning Scenario

2.2.2 E-Learning Platforms

E-learning standards consider mostly the format of the so-called learning objects, which means the learning content together with meta-data. The *Learning Objects Metadata (LOM)* standard considers exchange, search and use of such learning objects [48]. It can be understood as a layer between data base and application. The exact applications and protocols how learning objects can be manipulated are not considered in a technical sense. The *Shareable Content Object Reference Model (SCORM)* aims to standardize web-based learning. It is related to learning management systems and provides a structuring of learning objects to packages, a description of a run-time environment for the management of these packages and, in the newest version, guidelines for navigation through this content [157]. A collaborative streaming presentation would be, if properly described by learning meta-data, a learning object. Pausing or jumping can be considered as navigation procedures. Since group learning is not specifically considered in the standards, the effects of such navigations on groups would have to be handled externally.

Simple learning systems allow for courses to be enlisted in the system, and students may enroll for the participation at a course. The Blackboard system started with such a course system and has developed to an advanced suite with content management system, collaboration tools, wireless extension, and other features. Similarly, universal platforms like ILIAS provide for personal workplaces, where courses and modules as well as digital resources like books are shown [65]. For collaborative learning, discussion platforms and chats are used. In ILIAS, group systems also allow to form groups to share documents and discuss their contents.

Besides general learning platforms, specific systems that concentrate on (distributed) live lectures are installed, which also provide interactive services, e. g. the WILD system at the University of Mannheim [143]. Students have their mobile devices to participate in quizzes, to give feedback about speed or to ask questions on content. The set of slides, annotations and the audiovisual stream of the lecturer are not transmitted to the students' PDAs, but enhancements could also provide for this.

For the actual technical transmission of content, usual IETF protocols are used for downloading (HTTP, FTP) and MBone tools which run over RTP/UDP on IP multicast for streaming live

lectures. At some universities, dedicated hardware solutions for video-conferencing, mainly via ISDN, are used.

E-learning in universities is quite popular for cost saving reasons: Lectures and material for them can be transformed into digital documents quite easily, in contrast to the materials for more interactive teaching forms in schools. Some universities just make course material available for download, but general learning platforms gain more and more interest, since they allow for the integration of the production and management of documents, exercises and tests from different departments. Besides that, more interaction possibilities can be included, though these are today mostly restricted to off-line discussions or formation of workgroups after learning.

Collaborative streaming, if it is integrated into learning platforms, can foster interaction among students, because recorded course material can be re-played and worked out for a better understanding.

2.2.3 Requirements

Synchronization only needs to be strict (+/-80 ms) if devices are within audiovisual range [152], i.e. in one classroom. Devices inside a classroom are supposed to be interconnected by a low-delay network. Besides that, all client components are usually similar, such that devices and applications in this case should not differ (with respect to play-out buffer delays, e. g.). In cases of interconnected classrooms for distributed learning, the synchronization requirements are not strict. If audio discussion is used, an audio comment of a member that has already seen a certain presentation event should not reach the other members before this presentation event has been played out at their devices. This means that the synchronization skew must be smaller than the human reaction time plus the latency for the audio transmission of the member's comment. Psychological experiments found the simple reaction time to be about 220 ms [90]. This value can be taken as a lower boundary, because the effective amount of audio transmission latency may vary considerably dependent on the used hardware components and networking settings.

For learning environments, the management of groups is essential. The number of users within a collaborative group can be about 10 – 100 participants, where we suppose only about 20 to be in one classroom. Depending on the course system, explicit partitioning of groups can be useful. Besides, access control rules must be definable for the course, for example to allow pausing or not. Rules can be pre-defined for certain course types, but it must be possible to adapt rules and reaction of the group dependent on the course. System support for the supervisor at this task is desirable.

Since learners may reside in distributed network places, the transmission of control actions will possibly be delayed. This may lead to timing problems at the group management, which may in turn lead to conflicts or oscillating behavior. Thus, the group management entity must be able to handle several control requests for the same piece of data in a consistent fashion. For example, requests may be collected in a certain time-frame.

2.3 Spontaneous Meetings

Since wireless networks and mobile devices get more capabilities to transmit and display audiovisual media, streaming in wireless networks is an interesting service. People may be interested in watching the movies that friends are currently streaming, i. e. they want to copy sessions to

their devices by a few clicks. This application is shown in figure 2.7. A collaborative streaming service could support this need for session transfer, and providers could even offer lower prices for people who advertise movies to their friends.

Figure 2.7: Session Transfer in Mobile Scenario

People in such scenarios form a spontaneous group to get synchronized to specific movies and leave some time later, taking their movie with them. Here, a person joining a presentation may not want to be synchronized to the viewing position of the first person, but just start the movie at the beginning.

Another reasonable use case is to start a streaming session collaboratively and give people the possibility to leave the place and take the streaming session with them on their mobile devices, as depicted in figure 2.8. In this scenario, the possibility to dynamically leave a group and the independence of other people's group state are important because of changing networking conditions.

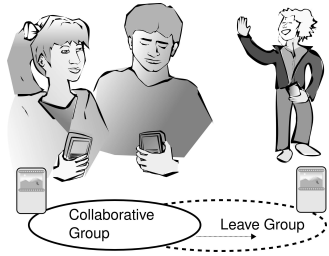


Figure 2.8: Leaving a Group

2.3.1 Wireless and Mobile Network Conditions

Wireless networks are either ad-hoc or infrastructure networks, whereby the latter can be cellular phone networks or WLANs at train stations or airports. In mobile ad-hoc networks without

infrastructure, users may want to send their video streams to others. However, streaming audiovisual presentations requires much processing power and battery performance on the sending end-system. Additionally, network conditions are subject to frequent changes, such that streaming in ad-hoc networks is less a case for movies, at least not for more than for small clips. Thus, specific characteristics and problems of ad-hoc networks are not considered further in this work.

In traditional cellular phone networks, usable bandwidth is relatively low, but the upcoming UMTS standard with HSDPA (High-Speed Downlink Packet Access) offers rates up to 14 Mbit/s [124]. Further improvements in modulation techniques even lead to higher bit rates in the near future. Since high resolution video is not needed if the movie is played on a small display, this is enough for streaming videos if efficient coding techniques are used. UMTS migrates from the usual channel switching of cellular networks to a fully packet switched network. Thus, data services like media streaming are a relevant use case for network providers to attract subscribers for the new UMTS networks. The *IP Multimedia Subsystem (IMS)* specification, which is available since Release 4 of the 3GPP standard, has been designed to handle multimedia services based on packet-switched networks. Internet protocols like the *Session Initiation Protocol (SIP)* are used to set up multimedia sessions [160]. A schematic overview of collaborative streaming in UMTS networks is shown in figure 2.9. Future developments of cellular networks, the so-called

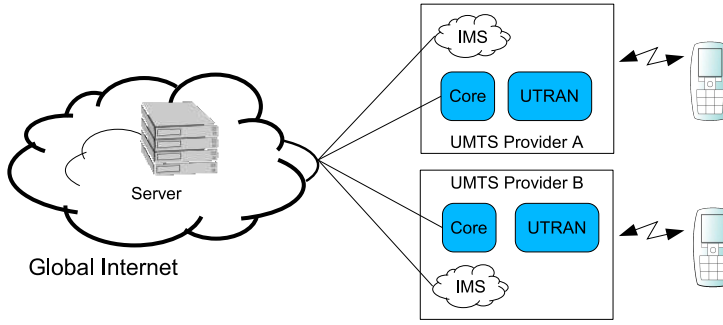


Figure 2.9: Overview of Streaming in UMTS Network

4G mobile networks, promise to integrate mobile phones with mobile Internet and offer bit rates up to 100 Mbit/s on the downlink.

WLANs at train stations, airports or other busy places in cities (so-called *hotspots*) technically operate like their in-home counterparts. However, users have to subscribe to the service, which is costly in many cases. Quality guarantees cannot be given, and continuous media services may be disrupted if the bandwidth used by traffic like file transfers is high.

Wireless network conditions are subject to changing signal conditions and possible disruptions when users move. Thus, packet loss is mostly caused by physical conditions instead of network congestion. The usable bandwidth is normally smaller than the given maximum values. In mobile networks like UMTS, QoS classes are provided which guarantee real-time data delivery. Additionally, advanced error correction and bandwidth adaptation techniques – which are outside

the scope of this work – can be used. We also state that media data upload will normally be impossible in mobile networks, because mobile networks are still not designed for this use.

For handover of mobile users, mechanisms inside the mobile network will be used, or network level mechanisms like Mobile IP may be implemented. Mobile IP allows a hand"-over that is fully transparent to the user, and transport connections may be kept up. IETF higher-layer signaling protocols are designed to allow breaks in the underlying transport connection, because they use a session identifier that is independent from the particular transport connection, so that the signaling state of the session or call may be recovered after a disruption.

2.3.2 Collaborative Streaming Requirements

Due to the possible mobility of users, as little state as possible should be set up in spontaneous meeting environments. However, a kind of registration must be made to indicate that a certain user is ready to invite others to collaborative streaming sessions or to receive sessions from others. Streaming session users will not always anticipate that they will meet others that are interested in collaborative streaming. Hence, a user group should not have to be defined in advance in order to transfer a session to joining users. In this scenario, session transfer is often initiated by the joining user, which is called a *Pull* transfer (cf. section 2.4.1). Since the set of involved users is generally unlimited, user and session lookup functionality play an important role in collaborative streaming in mobile networks.

Since it may be difficult to feed user addresses into mobile devices because of the restricted input interface of devices, an address location service should be offered. To achieve this, a user registration is necessary. If Pull services are to be offered, a registration of the available presentation is also required. These kinds of registrations can also be done after a streaming session has been started, since the information does not change continuously. However, in case a synchronization to the current streaming state is demanded, this synchronization information must be registered from the beginning, or it must be possible to query the state on demand (which is difficult with available implementations).

In spontaneous meetings, session control is not necessarily collaborative. This means, if each person controls the movie in an independent fashion, no access control is required. However, user groups that watch the movie collaboratively, like a student group on a train, may require aggregate control of all devices. Thus, the group has two possibilities: Either the whole group changes session parameters synchronously, or each user may change the parameters, like play-time position, arbitrarily, without any impact on other members. If any network disruption occurs such that a user “loses” the group, the second possibility can readily be used as a fall-back solution.

Since for spontaneous meetings, and above all if group members leave the place, the networking conditions may change and sessions may not be as persistent as in fixed network scenarios, data path sharing is normally impossible. Instead, the streaming service should offer to save state for disruptions due to bad networking conditions, so that the connection may be rebuilt if the network is available again.

In spontaneous group meetings, people probably will use different network access points with different service providers. Users somehow must be able to connect to a streaming server that offers the chosen content for them. This need not be the same streaming server for all group members. However, if different streaming servers are used, specific content discovery

mechanisms must be used, because a connection between the media server and the joining user cannot easily be made. Since such mechanisms are not available in a uniform way, it is assumed in this work that users either use the same streaming server or each streaming server offers the content at the same relative location. Thus, URLs may be easily rewritten with a different server address.

Another issue correlated with spontaneous, open environments is privacy. Sometimes, users would rather stream their presentations for themselves without being disturbed by joining users' queries. Furthermore, not everybody using a mobile device wants to be located and invited to collaborative streaming sessions. The discovery section of collaborative streaming configuration must handle this in an understandable fashion.

2.4 General Scenario

In the scenarios presented in the foregoing sections, several users watch a streamed media presentation together. Thus, a collaborative streaming group shares a common streaming session. Since collaborative streaming should not be restricted to static groups, different ways to add late joining members to a collaborative group must exist. These *session transfer* functions will be shown in the following subsection 2.4.1.

Sharing a common streaming session should not be restrictive in the sense that users always receive exactly the same media data. First, a heterogeneous set of devices and networks with different capabilities may participate in a collaborative streaming group. Therefore, it is reasonable to send media data to the users in such a way that clients receive individually adapted data. Second, users should be able to influence the course of a presentation individually, i. e. to *control the session*. Particularly, changing the position of play-time or changing the set of media tracks in a presentation should be allowed. These aspects will be discussed in subsection 2.4.2.

The different scenarios expose different group behaviors. At the time a collaborative group is established, certain parameters and rules are initiated. These rules affect access rights with respect to session transfer and control. Thus, a common group state is initiated at the time a collaborative group is established. How this state is managed will be described in further detail in section 4.2.

2.4.1 Initiation of a Collaborative Streaming Session

There are several possibilities to initiate a collaborative streaming session. First, the user may invite another one to watch a presentation, second, a user may join a presentation of another user, and third, a user may initiate a common group session before start of streaming. The two cases mentioned first are known as session transfer. Since session transfer can be used to make a usual streaming session collaborative and the initiation of a common group can use similar functionality, all cases are subsumed as collaborative streaming session initiation. The different use cases can be illustrated by UML Use-Case Diagrams.

The invitation of a joining user is shown as a *Push* function in figure 2.10. A distinction is made between the roles of the *Caller* who initiates the invitation and the user receiving the invitation, named *Callee* here. The Callee need not always be a person, but could also be a stand-alone client device, a case which has been explained further for home networking scenarios in section 2.1.

The Push functionality can be specialized as a *Copy* or a *Move*. With a Copy, both caller and callee keep the session and can control it, whereas the streaming service will be terminated at the

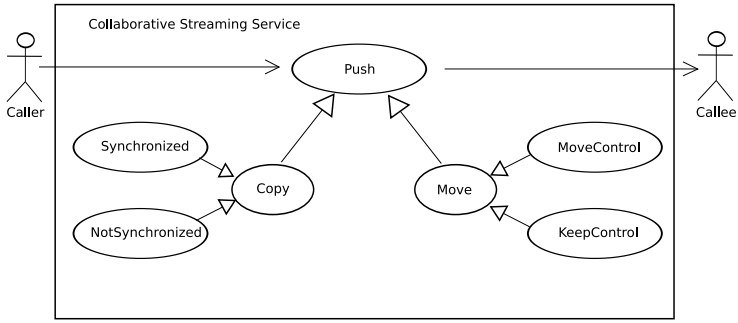


Figure 2.10: Push Use Case

caller for a Move. However, a specialized kind of Move lets the caller keep the control of the session. This can be useful for moving the session to devices without human user, for example. For the Copy functionality, specializations with or without group synchronization are available. Synchronization means adjusting the shared session state among the caller's callee's copy. In most cases, this means time-line synchronization, i. e. media data are played out at all client devices at the same point in time. However, other state variables like certain tracks may be synchronized also. In spontaneous meetings (see section 2.3), synchronization to other's session state may not always be desired. For the sake of clarity, the actual function for synchronization of users has been left out from the use case diagrams.

If a user wants to join a streaming session actively, the *Pull* functionality must be used as shown in figure 2.11. In contrast to the Push functionality mentioned above, the user roles are named

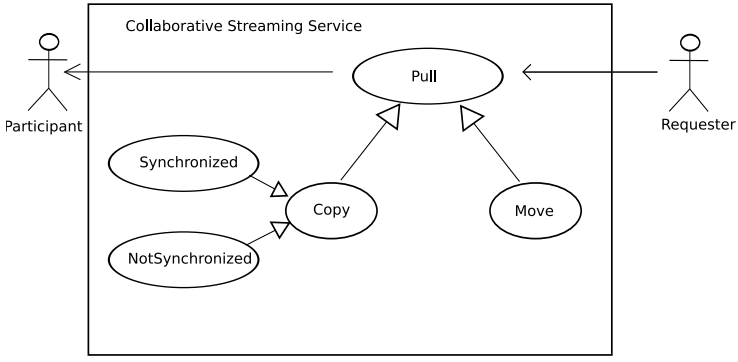


Figure 2.11: Pull Use Case

Requester for the user that wants to join the session and *Participant* for a user participating in the (collaborative) session. The latter need not be a person, but could also be a device that has no human user or a name that represents the collaborative group. Again, specialized cases like copying (with or without synchronization) or moving the session are conceivable. However,

moving a streaming session to the own device prescribes that the requester always has control abilities, therefore control does not have to be moved.

Starting a group for several users is handled by a function *StartGroup*, see figure 2.12. If

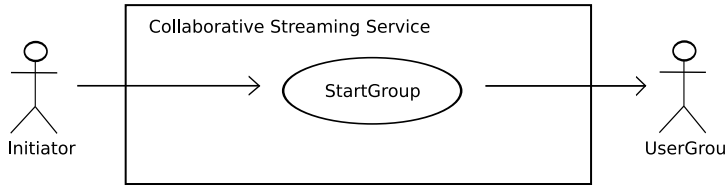


Figure 2.12: Start Group Use Case

this functionality is accessed, the *Initiator* of the collaborative session delivers the necessary information like group location or scheduled time to a group of users. The group is started in a synchronized fashion and the initiator is assumed to be able to control the session, thus the functionality works like a synchronized copy seen before, and specializations are unlikely to be useful.

The mentioned Push and Pull functions may be used to start collaborative sessions, but also to dynamically add members to collaborative sessions. Internally, the handling for adding members to collaborative sessions that are already set up and thus already have saved a common group state may differ from the cases where a new collaborative session must be initiated from an existing streaming session. In the latter case, parts of the streaming session state must be retrieved from the streaming system to form the common group state.

Each function invokes flows of interactive signaling messages, because users may deny access to the group state for a collaboratively streamed presentation or, on the other hand, may refuse the invitations. It is easy to understand that media data must not be sent without acceptance of the invited person. Besides that, to cooperate in a group actively, group state must be initiated and managed. The distribution of such state between networked nodes requires the use of a signaling protocol. Multimedia signaling protocols will be discussed in chapter 6 in general; their use in our architecture will be further examined in chapter 7.

2.4.2 Dynamic Aspects of a Collaborative Streaming Session

Users may control the presentation during runtime of a collaborative streaming session, for example by pausing or jumping to a different chapter. A use-case diagram for controlling presentations is shown in 2.13.

The roles of *Controller*, the client that requests an operation controlling a certain resource in a presentation, and *User Group*, which describes the rest of the group members, are distinguished here. Special cases of the *Control* use case are *Change Time-Line*, which is used to change the viewing position, and *Change Tracks*, by which a user may add or remove tracks from his / her streaming session. In contrast to usual streaming sessions, the access to control functionality in collaborative scenarios implies a change of the group state, represented by the *Update Group State* case in the diagram. If a group has to be partitioned due to management rules or specific

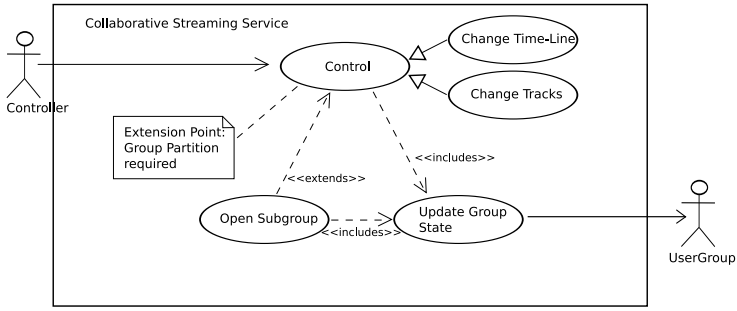


Figure 2.13: Controlling a Collaborative Streaming Presentation

requests, the *Open Subgroup* case is used to extend the control case. This in turn also implies an update of the group state, because the new subgroup will be initiated and the Controller will be added to it. If the group does not have to be partitioned, only the shared session state of the group must be updated. The UserGroup will then follow the action of the Controller.

Such dynamic behavior requires dynamic updates of the common group state. On the one hand, this affects membership, because users must be classified into subgroups dynamically, on the other hand, the shared session state of the subgroups must be updated. Media data, however, should be distributed independently to the clients for two reasons: First, individual adaptations because of client heterogeneity can be done more easily without a common distribution tree, and second, if a group has to be partitioned, the distribution of media data must be reconfigured if the group uses a common distribution tree. Nonetheless, parts of the data path may be shared in certain scenarios to transport media data as efficient as possible.

2.5 Goals of a Collaborative Streaming Architecture

Collaborative Media Streaming is a relatively new concept with little user experience until today. Different persons are involved who have different levels of knowledge and different quality concerns. Besides, mixed scenarios, and, with upcoming mobility of devices, handovers from mobile scenarios like spontaneous meetings to fixed scenarios like home environments are conceivable. Thus, finding fixed goals and design criteria is difficult.

Nonetheless, the analysis of scenarios shows that though there are differences, all scenarios use similar services, which may be handled by a common collaborative streaming architecture. However, flexible and extensible management is essential. Thus, the architecture can be enhanced to new scenarios and requirements easily.

We summarize the requirements of the different scenarios in the next subsection before we give some generalized assumptions and preconditions.

2.5.1 Requirements

The following requirements can be derived from the analysis of the different scenario types:

Session Transfer: The architecture must implement session transfer functionality as described in section 2.4.1.

A signaling protocol is needed to implement the communication between users and the collaborative streaming service. Since collaborative streaming should work in several scenarios, and media presentations are retrieved from Internet streaming servers, it is reasonable to base this protocol on IETF protocols. Signaling is usually done in the application layer. Since common IETF signaling protocols implement session state, these protocols can be considered as an implementation of a Session Layer.

Session Control: Collaborative session control is one of the basic characteristics of a comprehensive collaborative streaming service. As mentioned in the scenario descriptions and section 2.4.2, different situations require different access control patterns. For some reasons, people can require to split the group for a certain time. Group management mechanisms are required to guide such actions and reactions of a group. Especially, permissions for taking actions must be checked and the session state of the group must be updated, possibly by partitioning the group into subgroups.

Session Sharing: Clients that share a common collaborative streaming session must be synchronized to a common state. In most cases, this state means a common viewing position. The state must be maintained during the whole presentation, and can be updated on a collaborative session control command. Due to group partitioning, several subgroup states may be constructed and managed.

Inter-client synchronization must be strict in all environments where users may see or hear video and/or audio from two or more different devices. Experiments have found out that the synchronization skew for lip-synchronized audio/video should be within a range of +/- 80 ms to be satisfactory. Users tolerate skews even up to 160 ms, especially if video is played before audio [152]. The case for audio is more difficult, because the human audio perception is finer than visual perception. However, we assume that stereo speakers will be fed with one common audio signal. If users require different audio tracks, e. g. because of different language settings, those users would use earphones out of respect for others. Hence, the different audio tracks and the video track would have to be lip-synchronized, and acceptable limits of synchronization skew are the same as shown above.

Where user locations are distributed, the requirements for inter-client synchronization are less relevant, since any reaction on a specific play-time position will be delayed. For chat channels, this will never raise a problem, because reactions by humans almost always need seconds to be typed in and sent. In audiovisual back channels, the reaction of user A should not reach user B faster than the actual event. This condition will be fulfilled automatically if play-out buffers of the applications do not differ much, because coding on the audiovisual discussion channel also takes time.

Configuration: User-friendliness, especially in home networks, requires automatic device and networking configuration. Several standards (like UPnP) and research projects address these topics. In this work, only device and service discovery shall be considered. This will be restricted to local area networks, since this is sufficient in home networks as well as in learning environments. In these environments, device and service reconfigurations will also occur quite seldom. If users are connected to different provider networks in UMTS networks, specific services which take into account geographic location of users have to be used.

Not only devices and components of the collaborative streaming service, but also addresses of existing users and groups have to be located, because it may be annoying to type URLs for Push and Pull services. In spontaneous meetings, the privacy concerns of users should be respected.

Group Inter-working: An audiovisual or chat discussion channel can be useful (e. g. group discussions in learning environments) or not (family sitting in one room). Thus, such an audio or chat channel should be available optionally. Such a discussion channel needs resources, which can lead to congested networks in the worst case. If QoS mechanisms (like bandwidth guarantees by Integrated Services [12]) are available, the system should show if a discussion channel can be initiated without quality disruption.

In learning environments, discussion channels should correspond to the subgroups the members belong to, thus if a new subgroup is created, a new discussion channel is set up. In order to save resources, the discussion channel may be fixed in other environments.

Media Data Transport: Transport efficiency is a requirement that is important particularly for home and mobile networks. In home networks, techniques like caching can be useful, whereas buffer limitations for mobile devices require transcoding or layered coding techniques. If such techniques are not available, or user preferences (e. g. languages) differ, the streaming service must offer the setup of different tracks.

In heterogeneous networking environments, users are expected to access content from very different devices. Though mobile device hardware gets even more and more sophisticated, there will always be a gap between mobile and stationary device capabilities. In case of stationary devices, users expect to receive high quality media data, whereas in mobile computing, unnecessary media packets that add nothing to perceived quality should not be transmitted. Thus, provisions to choose from different quality levels have to be made. Again, transcoding or layered coding techniques could be used.

The quality levels to be used have to be signaled to the streaming subsystem of the collaborative architecture. Either the user has to select quality parameters on his own, or the system can try to find out the necessary parameters from the user's preferences or connection quality. In any case, the quality parameters of each user are independent of those requested by other users.

The collaborative streaming user application should also work for usual streaming sessions or streaming of live presentations and thus not be limited to stored presentations. That way, people can use the same application for many environments.

2.5.2 Assumptions

In this chapter, three very different scenarios with their environments have been analyzed. Even inside these environments, different set-ups occur. For the rest of this work, some assumptions are made that must hold for the collaborative streaming architecture to work properly. We start with assumptions required for the general streaming service.

In wide-area Internet environments, it is often recommended that streaming applications use a play-out buffer with a size of a few seconds to avoid decoder or display buffer underflow due to delay jitter. The resulting delay of such a large play-out buffer may be annoying for users, because it adds to start-up waiting time. Therefore, some applications allow to adapt the play-out

buffer size to the connection characteristics. Other solutions exist to shorten this waiting time by using a faster sending rate from the media server at the beginning of a presentation, and starting play-out when half of the buffer is filled. However, this adds to implementation complexity, so we will assume a play-out buffer of 1 second in a best-effort scenario.

Since the play-out buffer is by no means standardized, different applications may use a different play-out buffer size, which may lead to a deviation in inter-client synchronization. We assume that the size of the play-out buffer can be queried and distributed, though this is normally not the case in standard applications. However, without this assumption, inter-client synchronization is not achievable for arbitrary clients. Otherwise, the client applications must be synchronized using a specific synchronization protocol. This means all client applications must use a common clock, which is in the distributed case only possible by use of a common middleware. The buffers of client applications must be adaptable in that case.

It is assumed for simplicity that all applications and devices allow to run the IP suite over their networks, regardless of the networking standard they use. Further we expect that all devices in a home normally use the same connection for access to the global Internet. The case of choosing different connections for different streams of a collaboratively streamed presentation is not considered here, because even in case that a visitor is invited to join a presentation the home users are expected to share their home network rather than forcing the visitor to use an own, possibly expensive, connection like a cellular network. WLAN introduces the highest latency into a home network. However, WLAN latency is in the order of tens of milliseconds only. It is difficult to retrieve generally applicable latency values, but own measurements (using ICMP pings) suggest that the average round-trip delay of a WLAN is about 20 ms even in the presence of background TCP traffic. Fast Ethernet adds only little latency, which is below 10 ms even for large frames and larger network dimensions. Thus, we assume that round-trip networking delays inside the home network are less than 30 ms. Similar considerations hold for learning environments inside classrooms or local area networks. In arbitrary learning environments or spontaneous meetings, no delay bounds can be given, because the Internet is a best-effort network. A delay difference between clients which is larger than the size of the play-out buffer means that inter-client synchronization is not achievable. In the global Internet, overall latencies can vary from 10 ms over 100 ms to more than one second for some connections [74]. However, these measurements are taken for TCP, which is used for signaling only in media streaming environments. UDP latencies can be lower because retransmissions are not used. For our work, we assume that lip synchronization can only be achieved and is only required in local area networks, and thus leave the case of general networks with excessive networking delays to future work.

It is assumed that most groups are quite small (less than 10 users in home and mobile scenarios, less than 100 in learning environments), such that content distribution to large groups is not relevant here. Groups also mostly concentrate in local area networks. If different local area networks are used, as in the case of distributed learning centers, a common service that accounts for group state, like access control permissions, must be reachable.

2.6 Summary

In this chapter, we have described three scenarios in which collaborative streaming can be applied. In home scenarios, mainly the transfer to other devices and watching movies on several devices in

parallel are relevant. Collaborative streaming can be used in learning environments under control of a supervisor. A lecture presentation can be paused on demand of single users to ask questions. Subgroups on individual topics may be formed to allow small-group discussions. Finally, in spontaneous meeting scenarios a quick synchronization to a certain presentation is interesting. Here, the joining user has to find the presentation and the collaborative group.

All of those scenarios have their specific networking environments with certain technical conditions. In home scenarios, local area networks with relatively high bandwidth are used to connect devices, which are heterogeneous in nature. The connection to the Internet has less capacity in terms of bandwidth and delay. In learning scenarios, mostly powerful, homogeneous devices are interconnected by a high-bandwidth and low-delay network, whereas in mobile scenarios just the opposite is valid.

Resulting from the scenarios, use cases have been developed for collaborative streaming. Those use cases are applicable in all scenarios in general. Particularly, the initiation of a collaborative session and collaborative control during presentation play-out are important. The initiation of a collaborative streaming session results in an increase of the number of members. Initiation transactions comprise the start of a collaborative streaming session, a push to a new member, or a pull from an already joined member. Control during presentation means a change of the session state, which may result in a state change of the whole group, or in a partition into subgroups.

Finally, we have described requirements and assumptions for the development of our collaborative streaming architecture. Those requirements comprise the implementation of the above-mentioned use cases together with an effective synchronization of several client devices. It is expected that the synchronization deviation among two devices remains within satisfactory lip-synchronization bounds (+/-80 ms). The different challenges of groups with respect to session control have to be met by a flexible collaborative session control concept. Since groups in the presented scenarios have a limited number of members, a certain relationship among members is expected. Distribution of media data to large group is therefore out of scope of this thesis.

3. Related Work

Parts of the requirements for a collaborative streaming architecture have been implemented in commercially available applications or research projects. Session sharing has emerged as an application for intelligent home networking middleware. Additionally, other middleware standards like CORBA have defined extensions for multimedia data handling, and in several research projects, small middleware components have been designed to provide for efficient media data transport. These can also be used to implement streaming from a server to a group of receivers. However, these systems often use proprietary protocols, and, even more important, offer only limited collaborative session control functionality.

Different collaborative streaming approaches can be found based on the Internet protocol suite: Streaming from a server to a group of participants can be understood as multicast communication and thus can be handled by multicast routing and management protocols. However, these approaches lack any session control functionality. Recently, content distribution networks (CDNs) have emerged as another possibility to efficiently deliver streaming clients to several receivers. The Comodin project [40] uses such a CDN and enhances it by collaborative streaming functionality. However, collaborative session control is limited to the change of the shared state in that project also.

Collaborative control functionality has been addressed by Computer Supported Cooperative Work (CSCW) architectures. These deal with multimedia objects in a shared environment. However, the controlled objects mostly have discrete states. Additionally, objects like documents can be divided into more fine-granular sub-objects like pages, which can be controlled separately. Real-time streaming to a group of participants, which control the session collaboratively, is not handled in these architectures.

In this chapter, existing approaches for streaming to groups are discussed in the next section. We distinguish between multicast, peer-to-peer and content distribution network architectures. Afterwards, we present related middleware architectures and approaches for session sharing, which includes the dynamic addition of clients. Approaches for collaborative work are described in section 3.3. Tele-immersion projects as shortly described in section 3.3.1 often distinguish among global and local view, similar to shared and individual control of streaming sessions. Audiovisual conferences often handle session transfer primitives. Other topics in collaborative work are the access control provided by the group management and conflict resolution.

3.1 Group Streaming

Streaming media presentations to groups of participants can be used today already: the wide-area distribution of lectures or events is a well-known application. Since participants can join such

Internet streaming sessions dynamically, one could understand IP multicast streaming as the IP solution of session sharing.

Although in these scenarios, no relation between group participants is required, multicast approaches for group streaming are described here because of their relevance as data transport approaches. In research projects, providing quality guarantees inside such multicast streaming applications has been an issue.

3.1.1 Group Streaming using Multicast

Streaming to a group of users is an evident application of 1:n multicast, where a sender distributes data to a group of receivers at the same time. Thus, all receivers get the transmitted portion of data at about the same time, which is dependent on the network delay from sender to each receiver. Since this delay can vary and no assumptions are made about buffers in different receivers, the synchronization among a number of receivers is normally not as exact as in case a multimedia middleware is used. However, synchronization is often unnecessary for wide-area multicast streaming scenarios.

In IPv4, a specific address class, class D, has been reserved for multicast communication, whereas in IPv6 all addresses with a high-order octet of 0xFF are multicast addresses [30]. Thus, if a streaming server chooses an address in this range, receivers have to join this multicast group to receive their data. The management protocol IGMP provides for such join and leave requests [16]. Additionally, IGMP is responsible for setting correct states in the routers on the path between sender and receivers. IGMP is not sufficient for group management of a collaborative streaming session, since partitioning, changing group state or giving access control are topics which are not handled at all.

For efficient data transport with as little link stress as possible, specific routing protocols have been designed. All routing protocols, however, have their deficiencies dependent on the group scenario they are applied in. Recently, the PIM (Protocol Independent Multicast) architecture has emerged to overcome those deficiencies [1], [35]. In this architecture, optimized routing protocols can be used.

Since the server sends the same portion of media data to the group participants at a certain point in time, partitioning a group (for example to jump to a different viewing position) is quite difficult. A new group using a new multicast address would have to be set up. Available streaming server implementations thus do not support changes of the viewing position if multicast is used. Besides, individual qualities for heterogeneous receivers can be supported in a inflexible fashion only, namely by using layered coding techniques [99]. The individual quality layers are mapped to different groups, such that a client subscribes to more groups for a higher quality and vice versa. Not only does this require changes to server and client applications but also to the coding of the media presentation.

If multicast transport should be used for collaborative streaming with an explicit group management, some preconditions have to be fulfilled by client and server. First, the server must support that clients choose the play-time position, such that a new multicast group is set up for the new request. It is, however, difficult for the server to decide if the old multicast group should be kept up for other clients. Therefore, a group management must still be in place. This group management must also care about join operations of a client, where just the multicast address of the group to join to must be conveyed. The client must be able to change the multicast group.

This normally corresponds to a change of transport parameters, which must be supported by both server and clients.

Even if these conditions are met, IP multicast suffers from deficiencies in scalability, security, reliability and congestion control. For media streaming, reliability problems are less important, if end systems are considered, in contrast to data transport for caching which requires full reliability. However, scalability is a crucial issue considering a wide-area group communication architecture in the Internet. For collaborative streaming, there is a potentially large number of groups with a small number of participants. IP multicast has been designed for scalability concerning the size of a group, but scalability with respect to number of groups is critical.

Thus, it is reasonable to evaluate different proposals for efficient data transport without using IP multicast [33]. Overlay networks of unicast connections are currently discussed for end-system multicast, see also the peer-to-peer streaming approaches discussed in the next subsection. However, these approaches put more requirements on clients, since clients should relay data to others. Other proposals use specific intermediate systems to copy data, mostly as proxies on application layer. These systems are called *reflectors*.

Multicast to unicast reflectors have been used in several media streaming projects. For example, reflector tools for MBone sessions have been proposed, such as the MURS reflector, a part of the MECCANO project [85]. A commercially successful implementation of a reflector is Apple's QuickTime Broadcaster [4], which acts as a reflector for clients which are not capable of participating in multicast sessions. Both projects, however, are not intended to handle VCR-like control requests of clients to enable true media-on-demand sessions, which are important for collaborative streaming.

The REUNITE approach [153] uses a similar concept of reflection, but packets are copied on network layer by REUNITE-capable routers.

3.1.2 Peer-to-Peer Streaming

Often, the notion of collaborative streaming is confused with the concept of peer-to-peer streaming. Peer-to-peer streaming approaches have emerged from the concept of peer-to-peer networking, which has been designed to efficiently distribute large files in wide area networks with a large number of subscribers. Since these conditions are also given for video-on-demand systems, the approach can be used for download of multimedia presentations also.

Several research groups pointed out how peer-to-peer networking strategies could enable video conferencing or media streaming. In a simplified sense, an architecture is needed where all multicast functionality is implemented at the end systems. In order to forward packets to all group members, an overlay mesh or tree must be built with end systems as replication points. An example for such an overlay is shown in figure 3.1. The server has to distribute streaming data to the clients (or peers) P_1, P_2, and P_3. The overlay mesh building algorithm chooses P_1 as the first child to direct packets to. P_1 then sends received packets to P_2, which forwards them to P_3, in turn. Packets are sent by general unicast routing procedures on the respective paths visiting the intermediate routers (R).

If a client wants to jump in the streaming presentation, the overlay tree normally must be reconfigured, because parents of the client send data of a different viewing position than requested. Such a jump can be considered as leaving one tree and joining the other tree. However, children

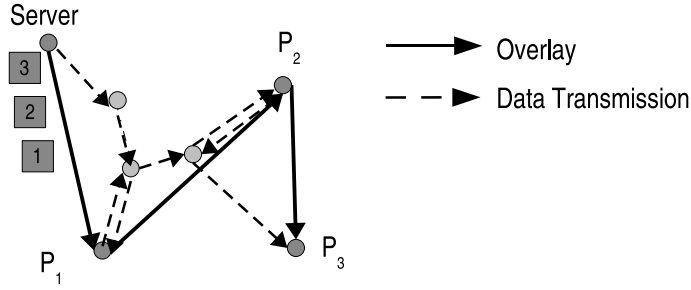


Figure 3.1: Peer-to-Peer Overlay Tree Example

of the jumping client are affected of the jump and must be connected to a different peer. An example of such a tree reconfiguration is shown in figure 3.2. Peer P_1 wants to jump to position 20 (corresponding to the sequence number here for simplicity), thus P_2 must be connected to the server itself to receive the media packets and forward them to P_3.

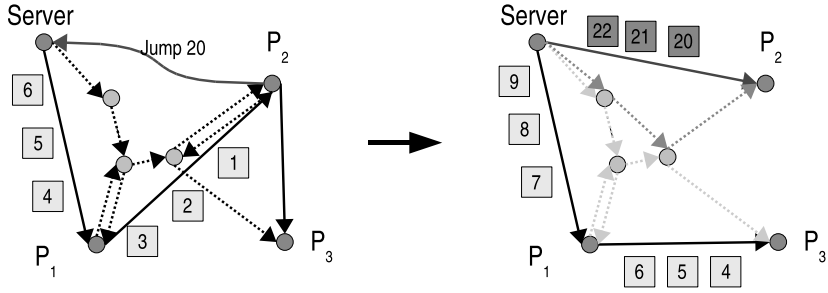


Figure 3.2: Reconfiguration of a Peer-to-Peer Overlay

The picture also shows a problem of tree reconfiguration in peer-to-peer streaming: P_2 has received packets 4–6 before the reconfiguration, but packets 7–9 have been received by P_1 only. The tree reconfiguration algorithm must ensure that P_1 still forwards those packets to P_2, but stops forwarding the packets with the higher sequence number. P_2 in turn must be prepared to receive packets from more than one sender in parallel. Since these operations require the interaction of the streaming application with the data transport, algorithms may get complex at this point.

In [22], an overlay tree building approach is presented which considers bandwidth and latency among group members to construct a reasonable routing structure for conferencing applications. The approach works quite well compared to other overlay building approaches and partially even better than normal IP unicast routing, because in IP routing some flaws exist. Since the authors use dynamic reconfiguration, joining a group can be mitigated. However, the approach needs considerable complexity at end systems for measurements, monitoring and tree building. This may not be useful in a variety of application scenarios which the collaborative streaming architecture should be deployed in, for example in spontaneous meetings.

In [76], peers in a collaborative environments store pieces of media data in their own cache to serve successive requests for a presentation. This is advantageous to reduce initial delay and delay jitter during play-out, however synchronous play-out at different clients is not supported in this architectures.

For streaming to a video conference, inter-client synchronization is a crucial issue. It is easy to see that in case of end system multicast some receivers will experience less delay than others. In the example before, P_3 receives packets later than both P_1 and P_2. Dependent on the effective networking delay, the clients' play-out buffers may have to be adapted to ensure synchronized play-out. If a peer with a low delay from the server joins the session, either this peer will become child of another peer and thus experience a larger waiting time until session start than necessary or tree rebuilding must be done, which may be as complex as shown above, because packets may have to be re-sent or receivers must combine packets from different receivers.

In peer-to-peer streaming systems, peers normally do not know each other, thus there is no need for group discussion and a common group state. In the collaborative streaming architecture used in this work, however, it is intended that users jointly watch a presentation, thus share a common group state.

Peer-to-peer streaming cannot be used as a mere replacement of collaborative streaming, since the high-level definition of a group is missing. The approaches target on scalable data transport in wide-area scenarios. Thus, a peer-to-peer streaming approach which handles tree reconfiguration fast could be used as the underlying data transport in scenarios, where inter-client synchronization requirements are less relevant. Nonetheless, client complexity is a disadvantage.

For collaborative streaming groups at smaller scales, i. e. in local or metropolitan area networks, the approach would mean too much overhead, and synchronization requirements may be too strict. Peer-to-peer streaming could in our case be used among intermediate proxies if groups are located in a wider area, so each local group could connect to a local proxy, and proxies build an overlay tree. However, since we expect to have sparse groups in a wide-area scale, we use unicast connections among proxies and do not consider end-system multicast in our approach.

3.1.3 Integrated Content Distribution Networks

An approach to integrate a content distribution network into collaborative streaming is followed by the Comodin project [40]. A content distribution network (CDN) denotes a server network that cooperate to deliver web or multimedia content to end users. A characteristic of a CDN is that client requests are directed to a server that is considered optimal for serving the client's request.

A number of CDNs have been designed and implemented for web or multimedia content. The VCDN (Video Content Distribution Network) offers content to clients by deploying a hybrid approach of CDN and peer-to-peer streaming [15]. A number of distributed proxies move or replicate content among each other and thus work like peers in peer-to-peer streaming scenarios. A central data base keeps information about current locations of content. This data base is queried by search services, which form the interface to VCDN for clients. Since proxies may be turned active or inactive dynamically, VCDN can be extended and decreased according to client load and thus utilize resources as well as possible.

Akamai is a commercial CDN operating on a private network [2]. The philosophy is to deliver content to clients from so-called edge servers, which are placed nearby the client's access network and replicate web or streaming content from the content provider's servers.

The research project Prism uses a number of so-called replay portals [26]. They are intended to be fed by so-called live sources, which are directly connected to content providers. The replay portal may archive the content and offer control functionality to clients. Thus, it provides for a network VCR (in contrast to recording video to equipment in the home network).

The CDN infrastructure of Comodin is based on the Prism architecture. Comodin is organized into two planes, the so-called *Base* plane and the *Collaborative* plane. The latter will be addressed in more detail in section 3.2.2. The Base plane is defined as a CDN platform. An origin server distributes media data to its *surrogates*. Each surrogate contains a *portal* for file transfer from the origin server, a *media streaming server*, which streams requested content to clients, and a local monitor agent for performance monitoring. Two control mechanisms are used by Comodin: first, the redirector which queries the surrogates for its performance measurements and decides about the optimal surrogate after a certain redirection algorithm; and second, the content manager, which controls content transfer to the surrogates and storage of media objects on the surrogates.

3.2 Approaches for Session Sharing

Multimedia session sharing is an issue mostly targeted to middleware, thus allowing for easier application development. Classical distributed object computing middleware like CORBA or JAVA RMI are considered less suitable for multimedia applications because of inefficient data copying techniques. Thus, streaming services have been specified for CORBA [158]. More flexible middleware architectures like Network-integrated Multimedia Middleware (NMM) as described below have been developed by research groups to integrate more heterogeneous device and transport requirements.

Session sharing approaches can be implemented on top of such middleware or independent from these architectures. In this section, first some underlying middleware concepts are presented before projects considering session sharing are reviewed.

3.2.1 Multimedia Middleware

Since traditional middleware has been optimized for request-response processing, streaming of continuous data causes high overhead due to expensive data copying at each request and inefficient data transformation. Thus, streaming extensions have been specified and implemented [110]. Creation strategies for streaming endpoints are decoupled from actual behavior. For actual streaming, various transport protocols can be chosen and used independently from earlier middleware data transport protocols.

The Java Media Framework (JMF) [154] can also be understood as a middleware due to its various pluggable device components which are integrated into abstract processor or player components. However, this approach is restrictive in the sense that applications may only access those components that are directly connected to them.

Some middleware approaches dedicated to the integration of multimedia devices in home networks have been presented in section 2.1.2.3 already. From these, we only present WWICE [6] in further detail, because it provides for specific services useful for collaborative streaming. For the

underlying device architecture and networking technology, WWICE uses similar concepts like HAVi. WWICE introduces the concept of an *activity*, which is the abstraction for a data flow from source to sink. A specific entity called *GraphMapper* maps activities to actual unidirectional data transmission graphs. These graphs are extensible and parts can be shared. WWICE is applicable in home networks only, but its follow-on project WWICE2 aims at connecting different home networks.

NMM (Network-integrated Multimedia Middleware) [94], a research project of the University of Saarbrücken, offers an architecture explicitly designed for controlling distributed devices and media processing entities. It uses the concept of abstract representation of software or hardware components as *nodes*. These nodes, as shown in figure 3.3 have input and output ports called *jacks*, which support certain multimedia data formats. Nodes are capable to store and process media data, which are transported in so-called buffer messages (B) and can handle event messages (E), either from other nodes or from the application, in parallel. Nodes should be fine-granular processing units. If more complex processing is required, they can be connected to so-called *active flow graphs* [94]. An abstraction of these flow graphs is a *session*, similar to the concept of activity in WWICE. NMM is well suitable for optimal use of in-home networking resources and fine-granular access to processing entities.

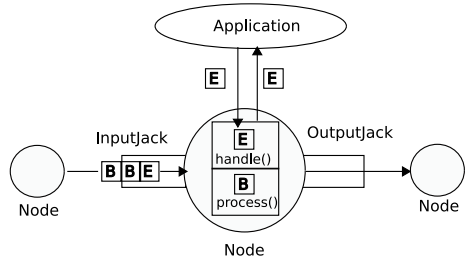


Figure 3.3: Data and Control Flow in NMM [94]

Similar concepts are described in the *Infopipe* model [10], where communication components are connected by ports. Exemplary Infopipe components are sources, sinks, buffers or filters. The model exposes high-level communication mechanisms, especially related to QoS, to application developers while hiding tasks like scheduling or thread management. Push and pull methods are used to invoke streaming communication.

The concept of nodes and active flow graphs of NMM is similar to the concept of *stream handlers* [47], where small media processing units (SH) are connected by a controller called graph manager as shown in figure 3.4. Thereby, reconfiguration of the data path can be done as appropriate. Each SH has one or several endpoints (EP) which are used as input or output ports. Data are not copied among SHs, only references to the data are passed. Hence, flexible distribution systems can be built, which support different transport protocols or efficient data delivery mechanisms.

3.2.2 Dynamic Addition of Clients

A research group of the University of Ottawa, Canada has proposed a late-joining approach with the JMF [172] which synchronizes clients to a common play-time position. The approach is

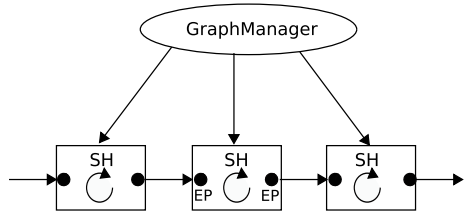


Figure 3.4: Data Path with Stream Handler Concept

intended for play-out of presentations in collaborative groupware. However, in this approach the play-out of the media presentation is interrupted at the moment a participant joins the session, which is not desirable in collaborative streaming.

WWICE allows to extend activities to other clients. This means that another sink node and possibly some decoder nodes must be added to the graph. Furthermore, WWICE offers copy and move functions by copying the nodes of a graph and replacing components of the graph, respectively. Synchronization of the sinks is not explicitly addressed. However, devices are expected to use the isochronous services of IEEE 1394.

Late-joining services have also been designed for other collaborative groupware systems. For example, the digital lecture board (dlb) uses a separate multicast group to deliver shared state information [45]. Latecomers send requests for current state to the usual session, whereon so-called late-join servers, which are chosen by an anycast mechanism, send replies to the separate group. At the university of Mannheim, an enhancement of this concept has been implemented by using an interactivity enhancement to RTP (RTP/I) [164]. In contrast to dlb, their approach uses a generic protocol. Furthermore, chosen late-join servers also join the separate group so that requests can also be sent to this group. In RTP/I, states and events of the medium are transmitted. Any application can thus use this protocol to query active state. The authors have also designed different late-join policies to achieve flexibility e. g. in case of lower network capacity. RTP/I has been specifically designed for the distributed interactive media class, which has a deterministic state at any given point in time. Audiovisual media streaming does not completely fit the characteristics of this class, whereas shared whiteboards are a typical representative.

A session sharing service implemented on top of NMM has been proposed in [95]. Parts of active flow graphs can be shared among several devices. In requests, nodes can be marked as shared or exclusive to express policies for sharing. On composition of the flow graph, the session sharing algorithm searches for overlaps to actually running sessions, considering the requested policies also. The authors propose a controller for global synchronization of these devices. Additionally, the graphs can be dynamically reconfigured to enable seamless handover in case of session transfer to different devices. However, no explicit group management is installed in this system, thus operations like pausing or jumping can be executed, but there is no entity that controls permissions or the effects of these operations on group state. Moreover, the required NMM components must be installed on each system. In order to use the proposed session sharing algorithm effectively, components must be defined as nodes with fine-granular processing units and give access to these nodes to other components. This is reasonable to be used in limited

scenarios as home networks for variable applications, however would mean too much overhead for larger networks or applications distributed on the global Internet.

The *collaborative* plane of the Comodin project [40] is shown in figure 3.5.

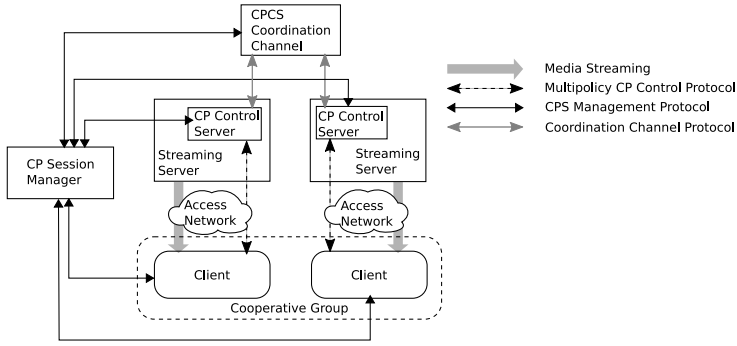


Figure 3.5: Collaborative Plane of Comodin Architecture [40]

Clients join a collaborative streaming session by sending a join command to the so-called *Collaborative Playback Session Manager (CPSM)*, which is the central entity for organizing a collaborative streaming group. The CPSM also delivers descriptions of media files and collaborative groups (denoted as Cooperative Playback Sessions (CPS) in Comodin). Additionally, a *Collaborative Playback Control Server (CPCS)* provides for shared session control. The CPCS is collocated with the media servers of the surrogates. Since the approach aims on large-scale content distribution rather than on flexibility, the collaborative session control is limited to a state change of the whole group. Specific inter-client synchronization is not addressed by this architecture, but the different media servers of one collaborative group are synchronized at session start and at each state change by the CPCS Coordination Channel. If the client that joins the group gets redirected to a new media server, this server is also synchronized. This approach also implements a group management, which will be described shortly in section 3.3.3.

3.3 Collaborative Work Approaches

In usual group streaming approaches, the concept of collaboration is neglected. Computer-Supported Cooperative Work (CSCW) uses a different approach. Groups are formed to explicitly share documents, i. e. jointly view and edit documents. Other collaborative work forms are audiovisual conferences and tele-immersion environments. The control of the common state in these environments follows certain access permissions, similar like session control in a collaborative streaming session. Furthermore, it is important to avoid and resolve conflicts. Another interesting parallel is the use of session transfer primitives to join or leave a session.

3.3.1 Tele-Immersion

Tele-immersion environments seek to combine virtual reality environments with audiovisual conferencing [91]. Thus, complex data sets that are presented to the user in a natural way can be

shared between users. Due to role specialization, the data set can be presented to an individual user in his / her specific fashion.

Some approaches try to apply the concept of multiple perspectives to such tele-immersion systems. The view on a virtual reality environment is classified into local and global perspective: In the local view, a user may act on the environment without change of the global state, whereas changes to the global view are committed and distributed to all users.

In [120], an approach is presented which allows users to apply visualization filters on complex data sets either locally or globally, in order to solve tasks collaboratively. Time is always globally shared in this approach. Experiments showed that users preferred localized views over global views. A number conflicts for toggling options and other update conflicts arose in their application. Hence, feedback channels are useful.

In tele-immersion environments, the data sets that people work on collaboratively are normally non-continuous data. Audiovisual data is used for conferencing. Time is shared globally, in contrast to a collaborative streaming application where it is reasonable to have different play-time positions.

3.3.2 Audiovisual Conferences

Audiovisual conferences have been implemented by companies to connect remote sites to work on common projects. In the past, telephones were used to interconnect people. During the last decade, video technology has advanced, thus dedicated telephone lines can be used for audiovisual conferencing, for example by connecting four ISDN lines.

In the Internet, the missing quality guarantees have been an obstacle for successful video-conferencing. However, more bandwidth can be provided today, thus video-conferencing works, as long as people accept drops in quality and partial disconnections. The ITU-T recommendation H.323 defines protocols for audiovisual communication on packet-switched networks [159]. H.323 is suitable for inter-working with the PSTN, since it is partially based on ISDN protocols. The so-called gatekeeper works as a telephone system with certain service attributes. Since several additional protocols are used for signaling, channel definition, authentication, or other attributes, the standard and its implementation are powerful, but also exhibit a certain overhead. Thus the Internet community developed the Session Initiation Protocol (SIP, confer section 6.2.2), which separates control from data transport. In order to let both H.323- and SIP-based applications inter-operate and to apply specific management operations, the Centralized Conferencing Framework (XCON, confer also section 6.2.1) has been developed.

An already mentioned problem in collaborative environments is the question who should be able to access shared resources. The so-called floor control protocols deal with this approach by granting an exclusive (in some cases, a shared) permission to access the resource for a certain time-frame. Several approaches for floor control have been developed, which can be classified according to several criteria [32]. First, the management can be centralized or distributed, second, the floor access can be random or scheduled. Specific mechanisms like token passing or certain policies can be used to determine who should get floor access. For the XCON framework, the Binary Floor Control Protocol (BFCP) has been developed [20]. In this protocol, a floor control server is used to manage the floor state and process floor control requests. Clients can request information about floors, available floor requests, other users, and the capabilities of the floor control server.

Another approach related to the access of shared resources is a so-called *social protocol*, which is commonly referred to as the standards of polite behavior. The idea is that users coordinate themselves instead of relying on system rules. For collaborative streaming, social protocols may be used as well. Since with the establishment of a conference, a text or audio discussion channel may be used, users may discuss the state and content of the presentation. In some scenarios, for example where users personally know each other, changing presentation state may be simply executed globally, because all group members accept this according to the social protocol.

3.3.3 Group Management in Streaming Architectures

The Comodin project already mentioned in section 3.2.2 provides for explicit group management operations. Clients can initiate cooperative streaming sessions and announce them to a data base, which can be queried by other clients. Those other clients can subscribe to the cooperative session and join it later. Different policy strategies are applied for the single session control operations. For example, each member can always pause the session, but has to hold a token for seeking inside a presentation [40]. Additionally, clients contend among each other about the successful execution of a session control request. Since each client belongs to a specific control server of the content distribution network, this contention has to be done among clients of this control server as well as among the contention winners of each control server.

In [101] a concept for so-called *group integrity* for multimedia multicasting has been proposed. Conditions for group integrity comprise requirements for group membership, member roles and group organization. Multimedia multicasting is understood as a n:m relationship of senders and receivers. The concept introduces three hierarchy levels. Meta groups consist of the participants of one multi-media multicast session. They can be divided into groups, which are defined as the number of subscribers to one media type of a presentation (e. g. , audio, video, or text). Groups may be divided into subgroups to integrate semantically identical data with different coding formats. Members thus can serve as so-called transformers that re-code data from one subgroup to another. A special kind of transformers are mixers that aggregate data from several sources into one output data stream.

Group integrity is formed by a set of conditions on subgroup, group, and meta-group level. These requirements describe valid states and state transitions regarding membership set, member roles, organization, topology and even data traffic. For example, a certain number of sources or the presence of transformer nodes can be required. Integrity is controlled by a central group manager entity according to a set of policies. These policies consist of action policies to re-establish integrity in case of violation and transition policies to handle potentially conflicting user requests. For definition of policies, a language called GML has been developed and presented in [100], since existing policy specification approaches have not fulfilled the requirements for specifying and controlling the mentioned integrity conditions.

For collaborative streaming, not all of the group integrity concepts have to be applied. Since we focus on a client-server model for media streaming, users will not act as sources regarding media presentation. We also assume that clients do not serve as transformers. Instead, member roles for session control actions, which are not addressed by this approach, have to be defined. Additionally, a subgroup has to be defined in a different sense, because the group partition has to be made in the case a client wants to execute a control operation individually. Media data streamed to different subgroups in collaborative streaming thus are normally semantically different.

3.3.4 Conflict Resolution

The aforementioned social protocols can also be applied in case of a conflict, since people in a discussion may talk about reasons for their behavior and develop a common solution. If social protocols cannot be applied, or system support for social interaction is desired, conflicts in a CSCW system should be resolved nonetheless.

In [106], coordination policies for resolving conflicts in a single desktop groupware system are presented. Those policies are classified according to the type of conflict and the initiative how a solution should be found. The conflict type may be global, i. e. affecting the whole application or workspace, related to a whole element or a sub-element. The forms of initiative are proactive if the owner or initiator determines the solution, reactive if the actions of other users are comprised, or mixed-initiative if information from all users involved in the conflict is related. The authors of this paper propose a variety of conflict resolution policies suitable for different types of conflict. For example, voting which is done by querying feedback with automatic resolution before a global change comes into effect is a policy suitable for larger groups also. Ranking of users is often used such that changes can only be applied if a user has higher privileges than another user. It may also be suitable to combine this policy with other strategies such that changes are only allowed if the initiator is privileged above others that are using a document.

For collaborative streaming, the distinction between global state and single elements is different from CSCW systems. Global state refers to time-line, single elements could be tracks of a stream. However, access to tracks of a stream will normally not affect other users, while changing the time-line will have an effect – either the group must be split or the play-time position of the whole group will be affected. Conflicts arising from this fact and possible solution strategies will be discussed in section 5.3.3.

3.4 Discussion and Comparison with Our Approach

We have seen that parts of collaborative streaming functionality have been implemented in different environments. Most of these projects or approaches are not designed as comprehensive collaborative streaming architectures, except the Comodin project, which has been developed parallel to the constitution of this work.

In the next subsection, we examine the characteristics of related approaches according to the requirements for a collaborative streaming architecture, which have been presented in section 2.5.1. Thereafter, we summarize the differences of the Comodin project to our architecture in section 3.4.1, before we conclude the chapter by presenting the contribution of our architecture in the context of collaborative streaming.

3.4.1 Characteristics of Related Approaches

In multicast or peer-to-peer streaming, session transfer functionality is not explicitly addressed. Clients cannot invite others to streaming sessions. Instead, the MBone environment has provisions to announce sessions in a session directory so that clients can join them. Since media data are distributed in a uniform way, sessions are shared, but there is no explicit inter-client synchronization. The inter-client synchronization is dependent on the different networking delays among server and clients. Shared session control is possible in general to change the state of the presentation. The necessary access control has to be provided by additional floor control

protocols. Individual control by partition of groups is not addressed. The possibility to determine members of a group is missing, since many multicast or P2P streaming projects address content distribution to large-scale groups where members do not know each other or are only loosely coupled. Hence, group inter-working is not a concern in these projects.

As a multimedia middleware for in-home networks, HAVi and projects like WWICE are targeted exclusively on home scenarios. HAVi does not implement session transfer primitives, but WWICE addresses copy or move functionality to devices. HAVi defines control for VCR functionality, but does not deal with the case that several receivers access this control. In WWICE, session sharing is possible, because their data flow graph can be extended. Generally, a central control device is assumed for session control, so that only one client at a time gains access to it. Due to the quality guarantees provided by IEEE 1394 channels synchronization boundaries can be met, even if they are not explicitly addressed. Groups are defined as device groups only. Hence, group partition according to the requirements of users is not addressed and individual session control is undefined. Audiovisual channels for discussions are possible in general, but are not related to the streaming session.

In CSCW, group inter-working is the primary concern. Session transfer is mostly implemented in a similar style as in the multicast scenarios, because users can join a certain announced group or access a certain shared document. Conferencing systems, however, provide for explicit mechanisms to invite users to a conference. CSCW systems do not address synchronization of real-time streaming sessions, but provide for a consistent state of shared documents. Since the definition of the group is tightly coupled with that shared document, access control functionality is implemented. Even individual state changes can be made if the unit under access control can be copied. A consistent shared document state can be achieved by merging and conflict resolution strategies [106]. In shared workspaces or video-conferencing, this state is restricted on policies for the floor management, which are given by the implemented floor control protocol. For the Group Integrity project [101], a well-defined notion of group state is used in combination with streaming. However, the architecture is built to support multicast delivery of live presentations and, therefore, does not consider control operations like pausing.

NMM offers a multimedia middleware architecture designed for control of distributed devices and fine-granular processing of media objects in a distributed system. For collaborative streaming, we relate to the specific session sharing mechanism of that architecture. Due to a dedicated synchronization controller which uses a global reference clock, session sharing can be handled well in this architecture. The session sharing algorithm of NMM tries to find overlapping flow graphs for media data distribution to several sinks. The algorithm also provides for a certain form of session transfer. Devices can be added and removed from shared flow graphs and handover to other nodes can be made. However, NMM uses a complex description language so that additional mechanisms would have to be implemented to simplify push or pull transactions for users. Shared session control is possible in NMM for all receiver nodes that use a shared source. Access control is implemented in NMM as sharing policy, which decides if a node has to be accessed exclusively or can be shared. This policy is valid for all requesting nodes, i. e. no distinction according to characteristics of the requesting node is made. Since the high-level definition of a group is missing in NMM, no individual control of a shared session which results in the partition of a group is addressed. Using an audiovisual channel for group discussion is generally possible, however mechanisms have to be added to integrate this discussion with the collaborative streaming session. NMM is therefore targeted on more general session sharing scenarios in environments where

each node can be equipped with the middleware. Single processing elements of media data distribution can be accessed and controlled. The intention of collaborative media streaming is different in the sense that a well-defined group of members desires shared access and control of a streaming presentation. The fine-granular access to processing resources is not a main concern.

The most comprehensive collaborative streaming architecture is Comodin, which has been designed in parallel to the development of this work. The Comodin project aims on providing shared session control to well-defined groups in a content distribution network (CDN). The authors mention learning environments as an example.

Comodin addresses all relevant aspects of collaborative streaming: It provides for session sharing and shared session control, implements a group discussion channel and defines a kind of access control. Additionally, media distribution is scalable to larger groups due to the use of a CDN. However, some issues we have found to be relevant for the use of collaborative streaming in different scenarios are not considered by the Comodin project.

The Comodin approach differs from the architecture developed in this thesis in the following aspects:

- The collaborative functionality of Comodin is tightly integrated with the media distribution. Explicit control components inside the CDN are designed, which manage shared session control and provide for session sharing by inter-operation with media servers. In our architecture, media distribution and collaborative functionality are separate. Architectural components are expected to reside within or near the access network of clients. Since clients of one group do not necessarily have to use the same media server in our architecture, a CDN can be integrated as future work.
- Comodin uses proprietary protocols for coordination of media servers of one group, management of a collaborative session, and session control. The protocol for session management is based on RTSP so as to integrate well-known streaming server implementations. As join and leave commands are added to this protocol, group and session management are handled together. In this work, we separate group management and streaming session setup and control, which allows for an implementation with the standard IETF signaling protocols SIP and RTSP.
- Comodin uses access control policies based on the different nature of session control methods. Additionally, a contention is performed among clients to resolve conflicts caused by simultaneous control requests. For the use of a collaborative streaming architecture in different scenarios, such an approach may not prove sufficient, because users assume different roles with different rights inside a collaborative streaming group. Hence, our system implements access control policies based on control methods and user roles, with the possibility to rely on social protocols for small-scale environments.
- Comodin does not address individual session control, but changes the state for the whole group always. No group partition is intended, due to the focusing on learning scenarios. In our architecture, groups can be partitioned, either by policy definition or explicit requests. Hence, more loosely coupled groups can also be supported.
- Comodin only implements session transfer similar to the “pull” use case. Clients look up announced groups and join them. Comodin does not define “move” transactions or the explicit invitation of a list of clients to the collaborative session.

3.4.2 Summary

The limitations of existing approaches can be summarized as follows:

- In multicast or peer-to-peer systems, inter-client synchronization is not explicitly addressed. Instead, it is assumed that the quasi-synchronous transmission of media packets is sufficient. Approaches that provide for inter-client synchronization like NMM are tailored to specific environments, where controllers can access all media processing resources like decoders or displays.
- Except in the WWICE project, no explicit session transfer primitives like copy, move or pull that are accessible to users are defined. Clients mostly have to discover streaming sessions on their own and join them with partially complex mechanisms.
- Streaming session control, if allowed at all, produces a state change of the streaming session for all clients. While this is desirable in some scenarios, there are applications in which an individual control of the streaming session is required. Collaborative work approaches do not handle streaming presentations as shared resources and thus do not address session control.
- A definition of a group with members is missing in almost all projects except Comodin and collaborative work approaches. Therefore, groups cannot discuss certain scenes on demand. Moreover, no access control policies which are relevant for shared session control are defined. Only the group integrity concept [101] provides for access control based on user roles, however this project does not consider session control of recorded presentations.

Hence, the contribution of this thesis is to develop a flexible and user-friendly collaborative streaming architecture, which addresses inter-client synchronization, but makes as little assumption on the accessibility of system resources (global clocks, decoder or display buffers) as possible. Synchronization of late-joining clients to a shared session must be possible for unicast streaming sessions also.

To simplify user access of collaborative streaming sessions, explicit session transfer primitives as shown in section 2.4.1 are designed. Session control is not restricted to changes of the shared state, but a streaming session can be controlled individually on demand. Such a collaborative control requires the definition which piece of the state has to be controlled jointly. Therefore, a group consists not only of the participants of the collaborative streaming session, but also of a policy that has to guide access control and the possible partition into subgroups. In the following chapter, these basic concepts for collaborative streaming are elaborated in more detail.

4. Basic Concepts

We have already seen the different facets of collaborative streaming in the reflection of existing streaming and conferencing systems and approaches to combine these. Before giving a general view of our costream architecture in the next chapter, we analyze the individual aspects required to be existent in a collaborative streaming architecture. Therefore, a definition of the most important concepts is given. Particularly the notion of a subgroup in our context, the so-called *association*, will be defined. This association serves as a central concept to implement both session sharing and session control.

To set up and manage a consistent group state is the task of the group management. It uses access control derived from the role-based access control model used in operating systems. As an enhancement to this model, specific reactions on session control actions are defined. These let the group stay in a consistent state.

Session control actions known from individual streaming are used in collaborative streaming also, however these actions can have specific effects on the group state in collaborative scenarios.

Finally, the practical implications of session sharing are described in further detail. Particularly sharing of a common viewing position, which means synchronized play-out on client devices, is important for a collaborative streaming session.

4.1 Definitions

As already mentioned, besides session sharing, no high-level definition of collaborative streaming has been given in existing work until today. In this section, formal definitions are presented to clarify the notions related to collaborative streaming.

A *streaming presentation* is any audiovisual presentation that can be streamed in real-time from a media server to one or several customers. A typical example of such a presentation is a movie. A presentation has several *attributes* like number and identification of media tracks, time range or static quality. These attributes are static and a meta-data description of them is given by the server. A recorded streaming presentation covers a certain period, which is also called *time range* of the presentation, with start and end times that restrict the searchable range. This time range is described by meta-data. The *viewing or play-time position* is a point of time within this range, expressing the time when the media unit at that position has been sampled, relative to the beginning of the presentation. The *content* of a presentation refers to the utilizable information for humans, for example the story line of a movie. Content can be represented by different media types, i. e. video, audio, textual information, and images. These media types may in turn be available in different occurrences with different values of their specific attributes, such as

different encoding formats or different languages. Such single media instances are called *tracks*, in protocol-related documents also referred to as *streams*. In this document, we use the first term, which originates from movie and media server technology. Note that the resolution of a track depends on the encoded format. Often, a track corresponds to a media instance that is meaningful even if it is consumed stand-alone without all other tracks. However, in more advanced audio/video encoding standards, a media instance can be split into several tracks (consider several stereo audio tracks of surround systems, or separate coding of background and foreground video). On the other hand, a full audio/video presentation can be multiplexed into one track, for example using the MPEG-1 system stream.

Available presentations are composed from single tracks either in a static or dynamic fashion, see also figure 4.1 for an exemplary presentation. Note that several tracks of one media type can occur in one presentation. In the static case shown in the left part of the figure, certain format representations are chosen for the presentations P_0 and P_1, and the user can only select from a restricted set of tracks, for example a certain language. In contrast, in a dynamic system the different media and format representations reside in a repository wherefrom clients can compose the presentation tailored to the users' needs.

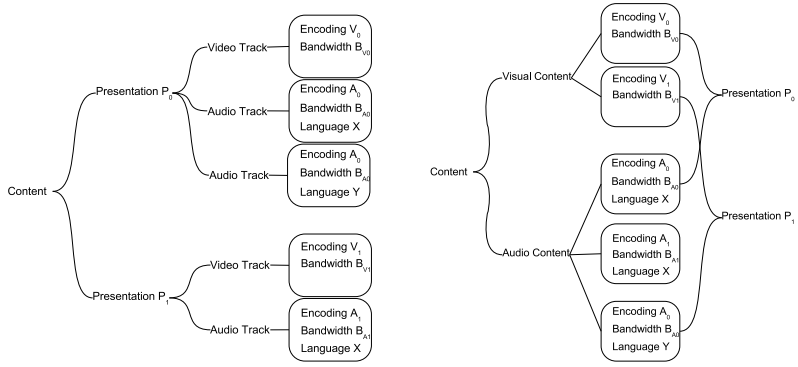


Figure 4.1: Content Representation Example

In available systems like DVDs or streaming servers, the first case is implemented typically. However, consumers of DVDs may often choose between a set of languages as audio tracks of a movie. Streaming servers often use a fixed correlation of one video and one audio track, thus the customer cannot select several different audio tracks and switch between them during the course of the presentation. Several presentations of the same content can hardly be related until today due to the lack of meta-data. Content always refers to one static presentation in our system. Thus, we also assume that group members watch exactly one presentation, which must then comprise all possible tracks. Clients have to be capable to select the relevant tracks. We state that in future, meta-data could be used to find equivalent presentations showing the same content. These presentations could even be composed dynamically. This would allow even more flexible scenarios of collaborative streaming. The basic concepts of our collaborative streaming

architecture are still applicable in this case, because the individual presentation management is separated from collaborative control.

In context of collaborative streaming, the notion of content is important for the group. Members of one group watch a certain content jointly. They may, for example, discuss the story or give comments, ask questions about the content, and so on. Certain representations of the content are mostly suitable for single users. For example, certain devices can only display suitable media formats, or a user wants to hear audio in a certain language. Mostly, these choices will not be relevant for the whole group. However, the group may also discuss about points related to a certain representation only, for example in case of a certain camera position in a sports TV coverage. Thus, group state need not be completely independent from the representation. For collaborative streaming, the notion of *sharing an attribute* is important. If members of a group share an attribute of a presentation, this means that its value is relevant for group partitioning as will be shown below. Sharing an attribute can be visualized as the case of discussing the content that is played related to a certain attribute of a presentation.

A *streaming session* comprises the state of a streaming presentation. This state can be described as the *session-specific attributes* like the current play-time position or the choice of tracks that are streamed. These attributes can change their values with time. The session is an entity which is managed at the streaming server. An identification of the session is given to the client. Thus, a session is independent of transport connections [146]. Typically, a streaming session refers to one client, however, if exactly the same state is shared among several clients, the session can relate to several clients. The latter case is practically implemented in multicast streaming and has the restriction that clients cannot easily change state.

This restriction is relaxed by the definition of a *collaborative streaming session*, which is the set of streaming sessions of the same presentation referred to a group of clients. The states of these sessions may differ among clients, however the group maintains a relationship among clients.

A *collaborative streaming group* (also abbreviated as *group*) is defined as the quadruple $G = \{P, C, CS, M\}$, with a presentation P, the set of clients $C = \{C_1, \ldots, C_n\}$, the collaborative session state CS, and the management policy M. The presentation P has a number of static attributes, which are announced by the streaming server and can be read and evaluated by members as well as by the group management. Examples are the time range of the presentation or information about available tracks. The client set C is changed by membership control operations like addition and removal of clients. The effective state of the client set is thus referred to as *membership state*. The *collaborative session state* CS is the union of the session states of C. Like in standard streaming sessions, the values of session-specific attributes are changed by session control operations. However, in collaborative streaming, the values can differ among clients of one collaborative group. The attributes that form CS are distinguished as follows:

- Shared attributes, denoted by $S = \{S_1, \ldots, S_k\}$, are considered for access control. A change of the value of a shared attribute can result in the partition of a group into subgroups as shown below. In practice, attributes that are considered as shared attributes represent semantically different parts of the presentation. Examples for this are the viewing position or certain tracks, say camera angles.
- Individual attributes, denoted by $I = \{I_1, \ldots, I_l\}$, are not considered for access control. Changes of the values of these attributes do not influence the structure of a group. Media

tracks that are only important for individual experience, like different audio track languages, are examples for attributes in this set.

- Certain attributes of S or I are so-called *informative* attributes. Their values are conveyed to other group members for their convenience. Informative attributes can be shared state attributes like play-time position or individual attributes like language settings.

For simplicity, we define attributes as a key-value pair. The assignment of values to the attributes may change for session-specific attributes, but remains static for presentation attributes. The distinction which attribute keys belong to S or I must be given in the *management policy* M, for the sake of brevity often referred to as *policy*. Client classifications and access permissions contribute to M. Optionally, constraints can be defined which generally restrict access, for example to enforce a certain group size limit.

Finally, a *collaborative streaming client* (or shorter *client*) is a client device with a collaborative streaming application running on it. A human user interacts with this collaborative streaming client and issue control operations related to the streaming session as well as to the user group. We have already mentioned that not every device has to be controlled by a human user, but a user could control several devices. Of course, several users may share (large-scale) devices using one or many remote controls.

4.1.1 Partitioning a Collaborative Group

The shared session state may be the same for the whole group during the streaming session, but to achieve flexibility, partitioning the group into subgroups that share a common session state must be possible.

An *association* is a subset of a collaborative group. It comprises the clients whose shared states are the same. All values of the shared attributes must be equivalent for all association members. A common example for an association state variable is the play-time of the presentation.

For session-specific attributes, i. e. the elements from S and I we define the session-specific value of an attribute a at client C_i as $a(C_i)$. The shared state at a certain client C_i is then $S(C_i) = S_1(C_i), S_2(C_i), \ldots, S_k(C_i)$. Accordingly, the individual state at a certain client C_i is $I(C_i) = I_1(C_i), I_2(C_i), \ldots, I_l(C_i)$.

Hence, we define the set of associations $A = \{A_1, \ldots, A_i\}$ with $i \leq n$ (n the number of clients) and each $A_j \subseteq C$. The condition that a client C_j is a member of an association A_k is defined as

$$C_j \in A_k \Leftrightarrow \forall C_l \in A_k : S(C_j) = S(C_l)$$

Thus, we map an association to a specific assignment of shared attributes. The association does not depend on the individual state of a client.

The set of associations is dynamic within a collaborative group, because clients may change the values of their shared attributes individually. As a consequence to these state changes, operations on the set of associations exist which create new associations or delete associations.

An example gives further insight to the definition: Consider a collaborative group with a set of four clients $\{A, B, C, D\}$ and the attributes given in table 4.1. The presentation has three tracks (t_1, t_2, t_3) and spans a time range of r_s until r_e. The management policy has defined

Presentation Attributes	Shared Attribute	Individual Attributes
Tracks t_1, t_2, t_3	Play-Time Position pt	Tracks
Time Range r_s, r_e		

Table 4.1: Attributes of Example Group

the play-time position as a shared attribute, whereas tracks are individual attributes that are not specifically controlled with respect to value changes.

Assumed that users A and B view the presentation at the play-time position 5:00 while users C and D view position 10:00. A, C, and D have selected both tracks t_1 and t_2, whereas B receives tracks t_1 and t_3. An overview of the group with its associations is shown in 4.2. A and B belong to one association; the clients C and D form the other association. It can be seen that the individual attributes are not relevant for the partition into associations.

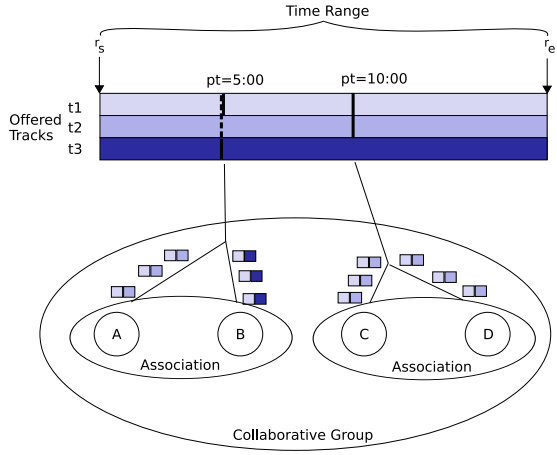


Figure 4.2: Overview of Associations of Example Group

A different setting, depicted in figure 4.3 could include selected tracks into the set of shared attributes, together with play-time position. This means that two clients form an association if both their play-time position and their selection of tracks agree. With the track selection given in the example above, A and B would each belong to an own association for their respective track set at play-time 5:00. C and D would form one association for their track set $\{t_1, t_2\}$ at play-time 10:00.

It is difficult to decide in general which partition into associations is applicable for a certain group. A reasonable approach could be to take only the semantically different tracks as shared attributes, whereas different format representations or languages are seen as individual attributes. Thus, only the selection of tracks that are shared attributes must agree for clients that are in the same association.

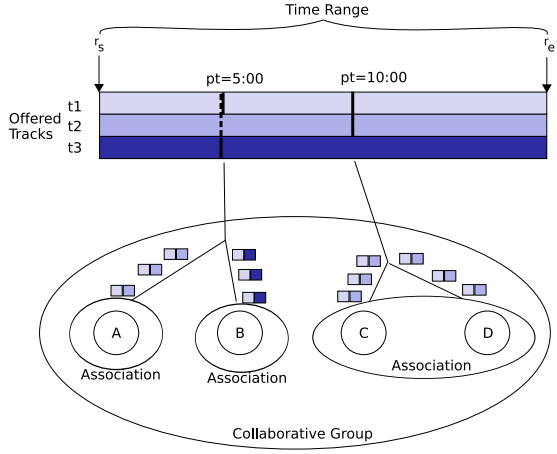


Figure 4.3: Associations Example with Tracks in Shared Attributes

With the definition of associations as subsets of the group's client set, we enhance the group definition as follows: The state of a group is defined as *valid*, if

1. in all associations A_C of the group G, the assignment of the shared state attributes is identical: $\forall A_C, \forall C_i, C_j \in A_C : S(C_i) = S(C_j)$
2. constraints in the management state are fulfilled.

4.1.2 Control Operations

Control operations are requests that influence the state of a collaborative group. Except for static presentation attributes, all kinds of state can be influenced. The differentiation by kinds of state leads to a classification of the control operations:

- Session transfer operations influence the set of clients, i. e. the membership of a group, by adding or removing clients. They are considered as membership control operations.
- Session control operations influence streaming session state. In collaborative groups, such an operation may change the values of shared attributes. These operations, which may differ among groups with the set of shared attributes, are called *shared session control operations*. In order to keep the state of the group valid, the system must react to such a control operation as will be described below.
- Management control operations influence management policy. The set of shared and individual attributes is changed by these operations, constraints can be added or removed, etc.

It is mandatory that the system checks if the requester of an operation is allowed to execute the operation. If this condition is fulfilled, the system must ensure that the result of the operation leads to a valid group state again.

Since the space of shared attributes is rather small, there will be a small number of shared session control operations. An evident example for such an operation is the pausing of a streaming session.

The system must react on this request by either pausing the stream for the whole association or put the requester into a new association.

Another type of session control operations are *individual session control operations*, which have implications on individual state only. They differ from all other operations in the sense that they may be executed at any time by any group member. An example for such an operation could be to enhance the quality of the individual streaming session by requesting an enhancement layer track. Individual control operations need not be classified any further. If other members should be informed of certain operations, such a classification may be reasonable, but we leave this for further consideration.

4.1.3 Overview of Definitions

A brief summary of the most important definitions for collaborative streaming as used in this thesis is shown in table 4.2

Term	Definition	Comment
Streaming Presentation	A/V Composition streamed from Server to Client(s)	Streaming in Real-Time: Continuous Transmission of Data, Processing on Client Side as Data arrives
(Presentation) Attribute	Property of Presentation	Examples: Track Identification, Time Range
Track	Media Instance of Presentation	Corresponds to a Stream, Media Type with specific semantic and encoding Parameters
Time Range	Period covered by Presentation	Only defined for recorded Presentations
Streaming Session	Procedure of streaming a Presentation, associated with unique Identifier and State	State corresponds to Values of session-specific Attributes
Play-time Position	Time corresponding to sampling time of Media Unit	Relative to Beginning
Collaborative Streaming Session	Set of Streaming Sessions of several Clients	Relationship of Clients mostly tightly coupled
Collaborative Streaming Group	$G = \{P, C, CS, M\}$	Consists of Presentation P, Client Set C, Collaborative Session State CS, and management policy M
Collaborative Session State	Union of Session States of Participants	Contains shared and individual Attributes
Management Policy	Definition of Shared Attributes and Access Control	Classification of Clients, Set of Permissions
Shared Attribute	Attribute under Access Control	Change of Value influences Group
Individual Attribute	Attribute without Access Control	Individually chosen by Client
Association	Subgroup with identical shared Attribute Values	Dynamic within Group
Session Transfer Operation	Operation that influences Client Set	Example: Invitation of Client
Session Control Operation	Operation that influences Streaming Session	Example: Pausing

Table 4.2: Collaborative Streaming Definitions

4.2 Group Management

The definition and classification of collaborative groups and their state is an important part of the collaborative streaming architecture. Additionally, group members may communicate directly with each other, for example in a chat or voice session. We state that group communication is already handled by a number of implementations (e. g. , ICQ [59] for chat and Skype [92] or Ekiga [141] for voice calls) and concentrate on mechanisms to manage a collaborative streaming session state.

Most collaborative streaming groups are dynamic (confer [98] for a definition of related terms), since members may join and leave the group during a presentation, if not prohibited by an administrator. The lifetime is restricted to the duration of the streaming presentation in most cases; permanent groups are not considered in this work. The group is considered as a closed group for inter-client communication, and only the streaming server sends presentation data to group members. The client set is probably heterogeneous in nature, consisting of clients with different capabilities, which is dependent on the scenario. Some participants have knowledge of the membership, thus the group is determinate. We do not handle streaming to groups with unknown members in our architecture, because this can be handled by multicast communication mechanisms. Like in general Internet telephony systems (e. g. Skype [92]), users are expected to register with a certain entity to express their (general) willingness to participate at collaborative streaming sessions. If users join a group – which they also may do at a later point in time – , they share a certain part of the state.

An explicit group management component that supervises the common group state as well as the individual state of members (the latter at least as far as the group is concerned) and allows or disallows control requests on behalf of a group leader is important because of different aspects: First, the group leader should be supported by the system in controlling members, because admitting every single request would not be feasible for larger groups. Second, the members themselves should get information on what they are allowed to do. Third, conflicts can be resolved in an easier way with the help of a group management.

In the following subsections, we first present the requirements for a group management before we describe each of these in further detail.

4.2.1 Access Control Models

A common group state requires access control, because it can be accessed by a number of different members. In the context of multi-user operating systems, different access control models have been developed [115]. The *Mandatory Access Control* has been used in security-critical systems, since the accessed object is associated with a security level and each subject that wants to access the object has to be provided with the according security level by administrative authorization. In contrast to this, *Discretionary Access Control* is based on the identity of subjects. The right to access a certain object can be passed from one subject to another.

Finally, *Role-Based Access Control (RBAC)* considers the operations that can be performed on a certain object instead of the object access itself [38]. A role in that context is a concept that binds a set of members of the group to a set of operations or transactions. Transactions here directly refer to the data item they operate on. The binding to the set of transactions is the difference to the concept of user groups in operating systems like UNIX [140], where a group contains the collection of members only and it is difficult to determine permissions of such a group. Thus,

for a role-based system both members and transactions of a role should be easy to determine. Besides, the set of transactions of a role should less often change than the set of members. This allows for simplified management of object access: An administrator does not have to determine each permission for every single user but only needs to classify the user as a certain role member with the appropriate permissions. This concept is very helpful in cases where only a limited set of different operations has to be performed.

Enhanced role-based concepts allow for the definition of hierarchies, where a role can inherit members and transactions from another role, as well as the definition of constraints, wherein advanced concepts like mutually disjoint roles or cardinality limitations can be expressed.

Parts of RBAC have been adopted by conferencing systems as will be shown in section 6.2. In these cases also the limited set of operations that do not change frequently and the possibly high number of users suggest the use of this access control model.

4.2.2 Requirements for Group Management

The Group Management component has to fulfill the following tasks:

- Specify Group Management Policy: On initiation of a collaborative group, its management policy must be constituted. This means that the control operations must be defined as permissions to certain member roles. Especially shared session control operations, possibly with shared attributes as parameters, must be defined from the set of session control operations. Policies for the reaction on shared session control operations have to be defined herein also. Besides, specific constraints to the group as the number of members or associations may be set up. Typically, administrators or group creators specify policies.
- Enforce Group State: Each control operation in a collaborative group must be validated if it conforms to the above-mentioned management policy and if the reaction on it finally leads to a valid group state. To support this, each joining member is allocated a role. Finally, conflicts that arise due to different interests of members or mere distributed networking delays must be resolved.
- Communicate decisions to members: The specification of the initial management policy as well as changes to this policy during session runtime must be sent from the administrator to the group management. The policy can also be distributed to group members. State changes resulting from certain control operations must be communicated to the group members.

4.2.3 Management Policy of a Group

The management policy of a group defines the space of users, attributes and control operations. It constitutes permissions and reaction policies to keep the group in a valid state in order to avoid conflicts as far as possible.

Not every step of a collaborative session has to be predefined. It is only necessary to define member roles, not to mention each user by name or address. Additionally, control operations that change shared session or group membership state are defined.

Those definitions are important to support the constitution of a set of permissions that denote if a user of a certain role is allowed to execute a specific control operation. Such permissions have to

be in place to guarantee a smooth process of the collaborative session. For example, some kinds of conflicts can be avoided. It must also be guaranteed that the collaborative session remains in a valid state after execution of a control operation.

4.2.3.1 Member Roles

In the collaborative group management, roles are needed to effectively control access rights. In larger groups it would not be possible to give access rights to each client separately. Roles become important with the definition of permissions, i. e. the transactions (or control operations, in our case) a certain role is allowed to execute. The assignment of roles and permissions should not change frequently according to RBAC [140]. Permissions only need to be defined for control operations that access shared attributes or membership state.

Most users will have a simple role of *participant* with certain default rights, and only a few outstanding members will be given roles with more powerful rights, such as *administrator* or *creator*. These roles are, among others, also defined in the centralized conferencing (XCON) data model [114]. At least one role should be allocated to each type of control operations.

4.2.3.2 Shared Attributes

Not all session-specific attributes are interesting for the whole group (e. g. in unicast sessions, transport parameters are not relevant for other users). Potentially interesting attributes (also refer to section 4.3) are the play-time position and the tracks a client will receive. In order to have simple types rather than lists and to make a distinction between single tracks, we define each single track as an attribute with a boolean value indicating whether a client streams this track or not.

State changes of shared attributes always cause a certain reaction, which will have to be distributed to other association members. It may also be reasonable to give information about such state changes to group members who are in different associations. In this case, the shared attribute is added to the set of *informative* attributes (cf. section 4.1). All shared state attributes can be informative attributes, which allows the distribution of state changes for the whole group.

All session-specific attributes that are not shared are individual attributes. These individual attributes do not contribute to the shared state. However, some of these attributes may also be informative attributes, in case their state changes are worth to be communicated to other members.

Privacy concerns may prohibit passing state change information, though streaming session state is normally no critical information. If users are tagged *anonymous*, no information about their session state is distributed to other members.

If session control operations access only one attribute, it may be easier and more efficient to directly describe shared session control operations as permissions to member roles. However, if a certain control operation has both shared and individual attributes as parameters, the shared attributes must be described in the policy also. For example, an operation which adds tracks must be parameterized with a track identifier. If some tracks contribute to shared state, whereas others do not, the respective track identifiers must be given as shared attributes.

4.2.3.3 Reaction Policies

We already have mentioned that shared session control operations must cause reactions of the affected association. Either the state change must be done for the whole association, or the user

requesting the state change must open a new association (or join to another association that already is in the requested state).

In order to decide automatically which reaction should be taken, the policy definition is enhanced with the so-called *reaction policy*. For each shared session control action, the choice between state change or opening a new association is defined. Additionally, conditions can be given under which circumstances a reaction should be taken, for example for a certain member role. For efficiency reasons, the requester may be forced to join an existing association that is in the requested state.

Problems may arise if reaction policies are used in combination with specific constraints to a group (confer also section 4.2.4.2). For example, if a new association is opened, it is not guaranteed that a constraint on the number of associations always holds. However, it is difficult to decide if reaction policies lead to invalid state beforehand. Thus, validity checks during runtime are needed as shown in the following subsection. If no constraints are given, a simple reaction policy would be to always open a new association on a state change.

4.2.4 Enforcement of Group State

During runtime of a collaborative streaming session, several control operations may be requested. Each of these may lead to a change of membership or collaborative streaming session state, and the group management must ensure that the whole group state is still valid. In the following subsections, we will first define how the management policy is initiated at all and how newly arriving members are equipped with the relevant parts of state. Next, the validity of control operations with respect to state must be checked using the rules of the reaction policy and the given management constraints. We will see that validity checking will be done in two stages as shown in figure 4.4: first, it is checked if the user is allowed to execute a control operation, second, the reaction on that operation must be evaluated whether it leads to a valid state. If no management constraints are given, the reaction validation can be left out.

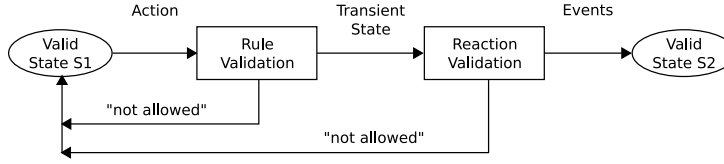


Figure 4.4: Validation of a State Transition

Even though this processing itself leads to valid states for each control operation, conflicts may occur because of different control requests arriving at short timescales. We show a few commonly known conflict resolution policies and give hints how they can be applied.

4.2.4.1 Permissions and Membership Initiation

Normally the user who initiates the group has administrative rights also; therefore this user has to specify the group state. This means that he or she must define the policies for membership state, collaborative session state, and management as defined in section 4.1. He or she certainly should be supported by the system.

The specification of the group state may be done beforehand for many parameters (e. g. the shared attribute play-time position), for example by editing policy documents. On the other hand, some parameters may be changed at the time of group initiation or even during the course of a collaborative streaming session. The specification thus must be conveyed to the group management.

Additionally, some attributes for collaborative session require to review the presentation description. This may happen in case that specific tracks have to be added to the shared attribute set. The presentation description can be retrieved from the streaming server. If the collaborative streaming service supports different streaming implementations, the administrator may also give the kind of implementation of the streaming system that should be used. Thus, a mapping of protocol requests to control operations can easily be done.

If a new user is invited to the group or joins it, the set of permissions can be delivered to the client so that it may display the permissions or even check the permissions before a control request is sent out to the group management subsystem.

4.2.4.2 State Validation

The notion of collaborative group state implies that such state should be valid for the lifetime of the collaborative group. There is one condition that must always be fulfilled to have a valid collaborative streaming session: All members of one association must be in the same shared state, i. e. their shared session-specific attributes must have the same values. This condition requires reaction policies. If a suitable reaction policy has been defined for each shared session control operation, this condition does not have to be checked additionally.

Validity constraints can be different for each collaborative group and may even be adjusted during the runtime of the streaming session. Such constraints can be defined for shared session state as well as for membership state. For individual session state, constraints should be defined individually to make sense, and validation of these constraints must be done in the client. An example could be that tracks may not exceed a certain bandwidth. In the rest of our work, we do not elaborate on such constraints any further.

Examples for constraints on membership state are restrictions on the number of members or associations. Constraint on session-specific state can mostly be directly mapped to constraints on actions, because these are used to change the session-specific state. An example for the latter is that only jumping forward is valid.

Thus, in the permission validation, a control operation is examined in the sense whether it violates explicit permissions and constraints on session-specific state. If an operation passes this stage, the rest of the constraints is checked. If any of these steps fail, the operation is not executed and the state reached before is kept.

4.2.4.3 Conflicts in Group Streaming

In collaborative environments, conflicts may arise due to the fact that several users access a certain element and try to change the state of it. For example, if changes to the time-line are made by several users in a short timescale, i. e. one tries to pause the presentation, another one wants to jump to a different position, the user demands cannot be fulfilled both.

If an attribute is not in shared state, a change of its value does not mean any conflict. For example, if the selection of tracks is considered as individual state, a user may stream an additional track

to his/her presentation without affecting others. Solely the additionally needed bandwidth may disturb others if a best-effort data transport service is used. However, problems with Quality of Services should not be handled by the Group Management and are not further examined here.

We denominate a conflict in collaborative streaming if two or more users explicitly execute a shared session control operation within a certain time-slot. In this case, it is assumed that users did not know about the other users demands and the two requests just collided on the group management. Another type of conflicts arises if users have different opinions, which may lead to oscillating behavior if all of them advocate their position.

Conflict resolution strategies may be applied on the action or reaction side of the group management. Mostly, the aim of such a strategy is to resolve conflicts at an early stage. In each case, a conflict may never lead to invalid group state. In order to be able to apply conflict resolution strategies despite networking delays, it is important to collect requests during a certain time-frame and then decide on the resolution. The time-frame should be in the order of hundreds of milliseconds, because this value allows to cope with different networking delays for most connections within the global Internet, but at the same time does not contribute too much latency on the reaction to a session control action.

The strategies mentioned below, which are derived from [106] and adapted to collaborative streaming, can be seen as additional constraints to operation permissions:

- Functional Overriding: A pause request could have more priority than a jump request.
- Ranking: Users can be ranked. Except for a group leader, who has priority over all other users, this strategy will not be used in our collaborative streaming scenarios.
- First request served: The request coming first will be served, other requests will only be served after a certain hold time. The user initiating the later-coming request is informed that his/her update was not carried out so that the user may either renounce sending the request, resend it, or leave the group forming an own association.

Reaction policies may also be designed so as to avoid conflicts as far as possible:

- Always split: Groups are split into single associations on each request. This policy is useful for spontaneous meetings, and could also be established in home environments. In a learning course where a group leader encourages free work and discussion it may also be useful.
- Always update group: This may lead to oscillations if malicious users want to obstruct the session. It may be applicable in home networks for people who rely on social protocols.
- Voting: Users are presented voting widgets to query feedback if the change should be applied. This policy can be useful in distributed learning environments. It is less useful in home environments and should not be used in spontaneous meetings.

Another strategy which is probably applied in many cases anyway is to let the group leader decide. This decision may be done in advance, for example by granting certain persons the right to update, or in reaction to a conflict. This is especially useful in learning scenarios.

4.2.5 Communication of Group Management and Clients

In this subsection, we show how the decisions of the group management can be communicated to clients. We also give definitions which part of group management must be communicated to clients at all.

For the communication of state changes, it is reasonable to use mechanisms of publish-subscribe architectures [36] as shown in figure 4.5. State changes of certain resources are published at a message *channel* which users subscribe to. In collaborative streaming, members of a group or association may subscribe to a channel used to inform about changes of state of the group or association, respectively. Initially and at each publication of a state change they will receive a notification.

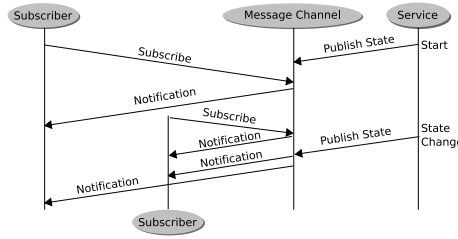


Figure 4.5: Subscription to and Notification of State and its Changes

Such architectures are advantageous in group communication because clients need not poll the state in regular intervals, and participants that generate events do not have to keep records of other members that should be notified.

4.2.5.1 Events

Events are sent out not only as a result of state changes, but can be useful also as informative means, for example if anybody has joined the group, information about the new member could be sent to the other members.

It is reasonable to have some predefined event types which one can choose from:

Session Control Events On reception of an event of this type, clients should update their own streaming state accordingly. There must be enough information in the event for the client application to perform the update.

Membership Events Events of this type are sent if the membership state itself changes, i. e. if new members join or leave, or members join different associations. Though updating a group according to the information given in this event is not as vital for the streaming session as it is with session control events, it is relevant for a consistent group state.

Informative Events Such events are not necessary for keeping a consistent group state, but they may contain interesting information for users which could be shown to them on demand or after a certain profile.

Administrative Events Such events are mostly relevant for the administrator, they could be sent if some user tries to access a function not allowed for his/her role. It may be reasonable to let only administrators subscribe to this kind of events.

The joining and leaving of users of a collaborative streaming group is an important event, similar to chat or conferencing sessions. In most cases, other users should be informed about these events, so they can start or stop interaction with the concerned users. Inside a streaming group,

joining or leaving of an association should also be distributed, in order to enable content-based interaction among users – users can only discuss certain scenes of a movie if they know who else is watching the presentation at that point in time.

4.2.5.2 Subscriptions

Using a publish-subscribe architecture has the advantage that users can subscribe to notifications of exactly the state they are interested in. Thus, not all state changes must be communicated to all members of a group.

Each client must subscribe to session control events. It is highly recommended that clients also subscribe to membership events. Users may choose freely if they want to subscribe to informative events also. As mentioned before, only administrators should be able to subscribe to administrative events.

Since a large number of notifications could be generated on subscribing to informative events, it should be considered which actions should generate informative events at all and which events a user should subscribe to. The first can be handled in the reaction policy list, whereas the latter must be handled in the client application.

4.2.5.3 Client Reaction on Notification

By its definition, a state-change event requires a reaction of the client application. A different viewing position or a new track that has been set up must be shown to the user.

Other event types normally do not require client reactions, except for administrative events. The latter may be used to automatically collect statistics about members and their actions.

If an automatic reaction is not necessary, the information contained in the event will just be shown to the user. If a new association is opened, the association may be shown on the user interface and the user may decide if he/she should join it. If a new member has joined the group, either for larger groups just the number of members or for smaller groups the name and the association the user belongs to are shown.

4.3 Streaming Session Control

In this section, those variables which can be controlled within a streaming session are analyzed. It has been mentioned already that these variables form the session-specific state of a streaming session. We will also examine which of these variables can contribute to shared session state and which are controlled individually.

This will be done mainly from the user perspective, describing how variables can be controlled in collaborative systems. We also examine the impact that a certain control request has on other participants of the collaborative streaming session.

4.3.1 Play-Time Positioning

The fact that users can choose the viewing position in a streaming session in a random way and media data is sent according to these user requests is an outstanding feature of a real-time streaming system.

The play-time position in a presentation can be manipulated by the following operations:

Pause: In a real-time streaming system, this operation fires a pause request to the server, which will also pause transferring the media. During transport of the pause request, the client application still receives media frames from the server. However, it should only buffer these frames without displaying them to let the human user perceive an immediate reaction on the pause operation.

(Fast) Forward/Rewind: The presentation is played with higher speed ($\star 2$, $\star 4$, ...) in the respective direction. However, this functionality is not broadly supported by client and server streaming applications.

Jump: In contrast to video recorders, computer applications can allow to drag a time slider to the desired position. If a higher-level description can be used, a graphical interface can show section headlines or keywords, which are in turn mapped to the corresponding play-time position.

In principle, participants of a collaborative streaming session should be able to choose the play-time position in the media presentation individually. However, the play-time position can also be seen as the most relevant part of shared state in a collaborative session. For group discussions, it is essential that the play-out at the same viewing position is synchronized, see also section 4.4.2. This means that the play-time position attribute contributes to association state.

Group members that wish to change the play-time position either change the state for the whole association they belong to or are grouped in a different association as presented in section 4.1.1. In the first case, the group must be notified about the state change in order to let client applications adapt their slider or position indicator, whereas in the second case, the notification of other group members is not necessary for their streaming applications, but is reasonable for the collaborative application.

If requests for a new viewing position lead to new associations, other participants may synchronize to this new play-time position. Besides, after some time, a group member may wish to re-synchronize to the main time-line. Thus, an explicit synchronization command to join a certain association is reasonable. In common streaming protocols, this multi-party control request does not exist.

4.3.2 Choice of Different Tracks

Exploiting different tracks is a very interesting task for a streaming system. With upcoming object-based media coding systems and richer description capabilities, this will even get more attention in the future. The streaming server can offer a rich set of tracks in the presentation description, from which the client will set up only a selection of relevant tracks.

Presentations contain tracks that differ in their *media type*, which means video, audio, or text, or in their *quality*, e. g. video streams in different coding formats. Besides that, different *languages* or even tracks with different *semantics*, e. g. additional camera positions or textual information, can be offered.

Whether different tracks should belong to the shared state or not cannot be decided a priori in a general manner. The fact that the same track is streamed to several devices does not necessarily mean that this track is in the shared state. If different qualities can be chosen, this is done according to the capabilities of the network and devices, which is always an individual decision.

Similar considerations hold for type and language of the media presentation. Thus, those different tracks normally belong to individual states only.

The case can be seen different for tracks with different semantics. As an example, participants may discuss a spectacular sports scene which is shown best by a specific camera. Thus such tracks could be taken into the shared state to enable associations concerned with those tracks. If a different camera position becomes interesting, association members could switch to this track or some members may open a new association. Since some applications offer to watch several video streams in parallel, a group member can be member of several associations, as already mentioned.

The technical problem for the decision which tracks belong into association state is that descriptions today do not offer high-level semantics to distinguish tracks according to the above definitions. The media type and format can be deduced from the presentation descriptions offered today, but language and semantics cannot be described by streaming systems available today. MPEG-7 descriptions [108] and the MPEG-21 framework [107], however, could be used in future to provide for better descriptions.

For this work we assume that a human administrator of a group can and will decide which track should contribute to shared session state. Still, choosing tracks for shared session state will not apply to many presentations, however, information about the chosen tracks may be interesting for other users as well, even if associations in a group do not depend on the choice of certain tracks.

4.3.3 Transport Parameters

In Internet-based real-time streaming systems, clients and servers negotiate transport parameters at session setup. This negotiation includes source and destination addresses and ports and the chosen media data transport protocol. Different from conferencing systems, where two peers reside on the same level, in real-time streaming systems the client can propose parameters, but the server can override this proposal.

In case of the destination address, the server may refuse to send data to a different address than which has been used in the signaling process. Control and data paths are not completely separate in this case. This is done for security reasons, since a client could maliciously overwhelm any host on the network with big data volumes. If authentication is used, setting the destination can be allowed.

A multicast address is also normally chosen by the server, because several clients may use the same data stream then. For ports to send to, the case is normally different, since the client is responsible to open the chosen ports, so the server accepts the choice the client has taken.

For unicast collaborative sessions, transport parameters are of no interest to other clients and thus are never shared. If collaborative sessions should be transported via multicast, the multicast destination address and port numbers are just coincident with the play-time position and thus are very convenient to be shared. Using multicast for collaborative sessions needs some changes to clients and server implementations as described in section 3.1.1.

4.4 Session Sharing in Associations

In this section, the effect of session sharing is discussed, i. e. the actions the system must take to enforce a common streaming session state at an association of clients, especially for a common

play-time position. We leave out multicast transport address here, because a common multicast transport address just means that group members share an association, be it related to time-line or to different tracks. The procedures that have to be run are similar to what is explained in the following subsections.

The collaborative streaming system must help the association members to achieve a common state. Since the streaming server should not be changed, an intermediate system must be in place to adapt the state among the clients. The effect of choosing the same play-time position should be that a certain portion of the media data is displayed at the same time at all clients in an association. This requires the synchronization of several media data streams at a number of sink devices.

Another aspect of sharing an association, which is partly related to group inter-working, is a common group discussion channel. We assume that a chat or voice communication system is able to open different channels within a conference session and thus do not address this issue in more detail.

4.4.1 Addition and Removal of Tracks

In case that a common session state consists of a certain set of tracks, all members of an association must select this set of tracks at the time they join this association. A notification document must then include the track identifiers that have to be chosen.

Since association membership can be changed during runtime of a presentation, the streaming system must support the addition and removal of tracks during presentation play-out. However, widespread streaming systems may require to select a certain set of tracks at the time of session setup, i. e. before the presentation is started. To allow changing the set of tracks during presentation play-out in such a case, there are two possible workarounds:

Single tracks per sessions: From the beginning, each track is set up in a single session with a unique identification. A presentation would then be composed of a number of sessions. Adding and removing tracks is then correspondent to setting up and tearing down sessions. However, synchronization of the single streaming sessions is difficult, because timestamps are chosen completely independent.

Session reset: If a track must be added to or removed from a session, the complete session is newly set up, thus a new track selection can be made. The old session is torn down. Since the set of tracks belongs to one session, synchronization of the different media tracks is given by the common time-line, but the play-time position at which the sessions are switched must be kept in the system.

A noteworthy issue is the fact that clients cannot be forced to add or remove tracks, at least concerning usually available streaming clients. Real-time streaming as understood today means that clients execute session and track setup [146], whereas enforcing a certain track set would mean servers had to do this setup.

4.4.2 Synchronization

In a presentation consisting of several tracks, these different media streams have to be synchronized to a presentation time-line. If streams are saved or transmitted separately, presentation timestamps have to be available in each stream to reconstruct the time-line of the single stream

(so-called *intra-stream synchronization*). The timestamps of several streams have to be related to a global clock so that these streams can be played out synchronously. This procedure is called *inter-stream synchronization*. Those timestamps are normally included at coding or packetization time.

For collaborative streaming, synchronization among several clients, i. e. the presentation of a sample or frame with an equivalent presentation time-stamp at two different client devices at the same time is a central issue. The samples or frames need not be of the same content track, for example one client could stream video while the second client is streaming audio and both should be lip-synchronized.

Only few approaches exist which explicitly consider inter-client synchronization. However, some inter-stream synchronization approaches are general enough to consider synchronization to different sinks. A selection of such approaches is discussed in the following subsection.

4.4.2.1 Approaches for Inter-Stream Synchronization

Inter-stream synchronization can be achieved by means of adaptive play-out synchronization as done in the Adaptive Synchronization Protocol (ASP) [139]. ASP can synchronize streams from different sources to a number of sinks. A global clock, achieved by means of e. g. the *Network Time Protocol (NTP)* [104], is used as a base for scheduling the media play-out at the sinks. One stream, selected by a certain policy, serves as the so-called master stream, whereas the slave streams are synchronized to the master stream. In ASP, each stream is assigned one source client and one sink client. Both client types can control the stream. A sink client is denoted *master* if it controls the master stream, else it is called *slave*. A central controller has to be used to guide start-up as well as the selection of the master. ASP thus consists of four sub-protocols:

Start-up initiates the transmission at the sources or the play-out process at the sinks, respectively. The central controller estimates the start-up times of each component from the transmission times and sends control messages with these times to the agents. The overall start-up latency must be greater than the maximum delay of all paths.

Buffer control is a local mechanism to keep the play-out buffer delay of the master stream inside a certain target area (between Lower and Upper Target Bound, abbreviated as LTB and UTB, respectively). If the buffer delay moves out of this area, the master stream release rate is regulated.

Master/slave synchronization is the mechanism to keep the slave streams synchronized to the master streams by communicating the adjustment of the master release rate to the slave sinks. At the slaves, Low Water Mark (LWM) and High Water Mark (HWM) bounds are used to keep the play-out delay inside the critical area. If the delay moves out of this area, master switching as explained below becomes necessary.

Master switching is initiated by slaves sending "tentative master" requests, together with their rate adjustment and an adaptation request. The goal is to move all agents into the critical area again. It is important to execute all adaptation messages in the same order to keep synchronization. However, alternating requests for adjustments at the lower or higher bound can lead to oscillating behavior. Thus, priorities are used, and only one side of the buffer should be adjusted.

ASP proposes some selectable policies: The minimum delay policies sets LTB equal to LWM, and the node with the currently highest transmission delay becomes master. UTB is set to LWM

plus the jitter of the play-out buffer delay. In contrast, the minimum loss policy sets UTB to HWM and the node with the minimum transfer delay becomes master.

Other approaches for synchronization include event-counting where each packet is classified as in-time, buffered or late [171]. This approach equalizes the delay of the incoming packets to perform intra- or inter-stream synchronization. The play-out clocks can be re-calibrated if certain thresholds concerning the number of late or buffered events are hit.

The synchronization method of NMM also uses delay equalization but with a central controller [95]. This controller calculates a virtual delay out of all presentation delays the sinks submit to this entity. The controllers at the sinks then display the media data portion at the time the controller schedules as a target time. The stream with the highest synchronization requirements (mostly audio) is chosen as the stream other streams synchronize to.

Local play-out adaptation by pausing or skipping frames in combination with a target presentation time distributed by the server is another possibility to achieve synchronization among clients [147]. In this approach, receivers deliver information about play-out time of the last media data portion using a modified RTCP receiver report (confer to section 6.3.3 for a description of RTCP). The server synchronizes all receivers to one specific master receiver and sends a so-called action packet containing the target presentation time of the next media unit. The action packet is an RTCP APP packet. Receivers then correct their play-out accordingly.

Usually, all approaches for synchronization among distributed clients require globally synchronized clocks. Additionally, in multicast scenarios each media unit is distributed to all participants with the same timestamp related to the start of the presentation. In collaborative streaming scenarios, the problem for dynamically joining clients is to find an initial synchronization point. In learning scenarios, a central entity can copy packets to a dynamically changing receiver set. Inter-client synchronization can then be achieved using one of the approaches presented above. However, in case that two separate unicast sessions must be set up and one client is joining the collaborative streaming session at a later time, an initial synchronization point must be calculated in such a way that viewing positions at both clients are close together.

Another point is that not all clients have the same capabilities concerning clock synchronization. Again, in small world scenarios like home networks, a synchronization protocol may be offered by a multimedia middleware. In other scenarios, clients may not implement clock synchronization (e. g. , for administrative reasons) or cannot measure delays.

Thus, we propose to use different synchronization modes, which are adapted to the needs of the clients:

Reflected: An intermediate transport service called *reflector* copies packets to the clients. Any joining client is added to the appropriate reflector session, which corresponds to an association. The reflector allows jumping individually or with the whole group and is able to reconfigure the reflector sessions dynamically.

Independent: The intermediate system calculates the play-time position of the association the client wants to join. An independent streaming session is set up for the joining client.

In both modes, presentation timestamps are still used to enable locally synchronous play-out.

The collaborative streaming system will check according to scenario preferences which synchronization mode can be taken. At a session setup, it checks which association should be used and

gives back the play-time state of the association. For the reflected mode, this play-time state corresponds to an existing reflector session. In the other modes, this play-time state must be calculated as the current viewing position of the association.

In the following subsections, for each of both approaches the following aspects are analyzed in greater detail:

Initial synchronization: operations and calculations that have to be carried out by the synchronization procedure at the time a client joins the association.

Association state change: If a whole association changes the viewing position, synchronization has to be kept up regardless of the transport mode that is used.

Individual position change: If an individual client changes the viewing position, the client leaves the association and either opens a new one or joins a different one.

Optimizations: The basic approaches presented here aim at the synchronization of late joining clients to associations and the dynamic re-synchronization of associations. In order to provide for a more fine-granular synchronization during playtime, the concept can be optimized taking procedures from synchronization approaches presented above.

4.4.2.2 Reflected Mode

The *reflected* synchronization mode needs an intermediate system that is able to handle data packets as well as the control traffic. This system, also called *reflector*, copies the packets sent by the server and sends them to all clients in an association as shown in figure 4.6. Delays from reflector to clients may be different, thus synchronization is similar as in case of multicast transport. In order to support session control operations the reflector must be able to reconfigure the data path for each client on demand. Therefore, the reflector manages so-called reflector sessions which control both server-side and client-side part of the data path. Several clients in the same association thus can use a common server-side part of the data path and have their own client-side part each [78].

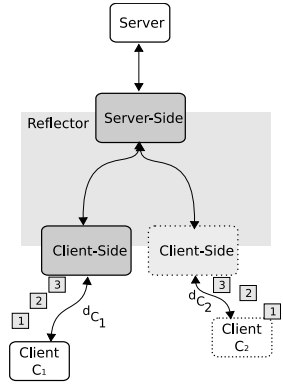
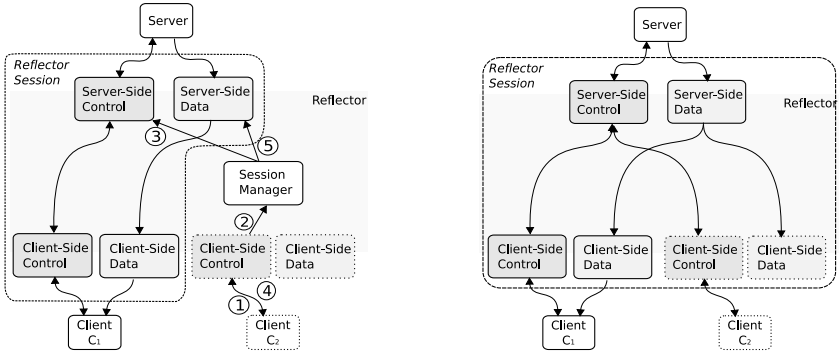


Figure 4.6: Reflector Copying Packets

The initial processing of synchronization to a certain (existing) association is shown in figure 4.7(a): After a request of an additional client (1), an existing reflector session corresponding to the requested association is located (2). The streaming control then does not have to interact with the server, but establishes an internal control connection (3) and sends own responses to the client requests containing negotiation information of transport parameters and play-time position (4). The reflector also connects the client-side data path to the already existing server-side data path (5) so that packets are sent out nearly simultaneously. The result of this operation is shown in figure 4.7(b).



(a) Control Processing (b) Result

Figure 4.7: Initial Synchronization in Reflected Mode

The approach can be coupled with multicast delivery on client side if this is available to save bandwidth. However, this is not recommended for today's networks, because multicast requires changes in client applications. Furthermore, multicast is only available in local area networks, where bandwidth constraints are less restrictive.

Changing the play-time position *for all clients* in an association, i. e. in a reflector session, is quite easy: The reflector just passes the repositioning request that it gets from the client to the server. The server may change the play-time position then and send the appropriate data packets, which are just copied to the clients. A little pitfall, however, is the fact that the control state is only delivered to the client who sent the request due to the request-response based control protocol. Thus, control state to the other clients must be delivered by other means, either by setting the play-time position as a parameter in the client or deliver an event containing the changed viewing position. The approach mentioned first has the advantage that the position change can be communicated by means of the control protocol itself, however not every streaming client supports setting parameters. The second approach is advantageous if the streaming client can be built in a modular fashion: A standard streaming client can be used in combination with an event interface to account for changes in the collaborative state. A graphical user interface can integrate the whole collaborative functionality.

For an *individual position change*, a lookup for an existing reflector session corresponding to the new play-time is done. If such a session exists, the client is added to this session as described above. If no such reflector session exists, a new session to the server must be opened. In terms of

streaming control, a resume or jump after a pause must be mapped to a setup to the origin server and the request to play at the new position. This will logically open a new reflector session the client is added to. The old reflector session, i. e. the server-side control and data path, will be deleted if no other client uses that reflector session any more. Otherwise, the respective client will just be deleted from the list of clients which use that reflector session.

The re-synchronization to an association must be supported by the group management if the state of the association is not known to the client itself.

Considering synchronization, the reflected mode works similar to IP multicast communication. However, the packets there are copied on the network layer at a router that has been selected by the routing and management protocols in contrast to this approach where a dedicated reflector copies packets on application layer. Thus, it is important that the reflector is placed nearby the clients in order to be bandwidth efficient and to avoid jitter. In local learning centers and home network scenarios, this is easy to accomplish. In distributed learning centers, there should be one reflector for each local area network.

The advantages of using a reflector instead of IP multicast communication is the support of flexible session control. Group partitioning is enabled using several reflector sessions. Since the whole collaborative streaming session looks like a usual unicast streaming session for a client, no changes to client implementations are needed, in contrast to switching between different multicast groups, which is poorly supported by existing servers and clients.

The disadvantages are that synchronization does not specifically consider the network delay between client and reflector. However, this can be mitigated by optimizations as shown below. Besides, different client capabilities like different play-out buffers are not naturally considered. This must be handled by a more sophisticated synchronization protocol, which can, however, inter-operate with the reflector. A more functional disadvantage is that changes in streaming state are not explicitly signaled to all clients in an association. This must be done using events or setting parameters in the client application. The handling of individual jumps is complicating the implementation. Another deficiency can be seen in the fact that the reflector session must be built at the time the first client opens the streaming session. However, with an implementation that effectively separates client-side from server-side path this can be easily accomplished in the reflector. Thus, only a mapping from reflector session to association state must be kept in the group management.

The approach can be optimized in two forms: First, the reflector could measure the delay to each client and introduce an artificial delay for each packet corresponding to the difference of the delay of the respective client to the largest delay experienced. Second, if global clocks can be used, one of the approaches presented in section 4.4.2.1 can be adopted: The reflector can distribute a target presentation time after collecting client feedback about their presentation times, similar to the proposal in [147].

4.4.2.3 Independent Mode

The independent mode is based on calculation of the current viewing position. At the start or at a state change of the collaborative session, the start time and position of the association are saved. For each client, an individual streaming session is initiated. The session of the joining client is then played from the calculated viewing position.

The approach does not need other prerequisites than saving the start time and position of each association in the proxy. As such, it is independent from other clients' behavior. Besides, no data path sharing is needed, because the approach uses functionality of streaming session control only. An overview of the control and data sessions of two clients in the independent synchronization mode is shown in figure 4.8.

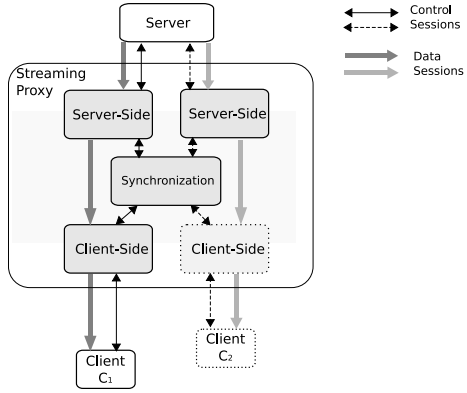


Figure 4.8: Independent Synchronization Mode

For the initial synchronization at the time a client joins the association, the association start time and play-time position is looked up. The viewing position is then calculated as the difference between start time and current time, plus the start position of the first client. The collaborative streaming system sends the viewing position to the server acting for the client.

An overview of the synchronization is shown in figure 4.9. The SAVE method saves start time (T_{Start}) and position (P_{Start}) at the time of the request arrival at the association service. After a certain delay d_1, which is composed of the round-trip delay to the server (including processing time) D_S, and the one-way delay from association service to client D_{PC_1}, the first media data of position P_{Start} are played out at client C_1. In reality, buffering delays at the client have to be added, but we assume each client has the same play-out buffer delay.

Later, C_2 joins at time $T_{Current}$. The SYNC method calculates the viewing position as $P_{Current} = P_{Start} + T_{Current} - T_{Start}$ [161]. We assume that the delay d_2 is different from d_1, in this case it is larger. Media data of the current viewing position are played out at a joining client C_2 at time $T_{C2} = T_{Current} + d_2$. At this time, client C1 already plays out $P_{Current} + (d_2 - d_1)$, assumed that its play-out buffer is filled enough to enable smooth play-out.

The problem is similar to what can be experienced in the reflected mode, however due to the setup of different unicast sessions the delay includes the delay to the server, which may have changed in the meantime. Thus, the calculation fault can be higher if the optimizations shown below are not applied.

Repositioning of *all clients* in an association is not as easy as in the reflected mode. Each client may jump individually, but if other clients are members of the association, they have to be

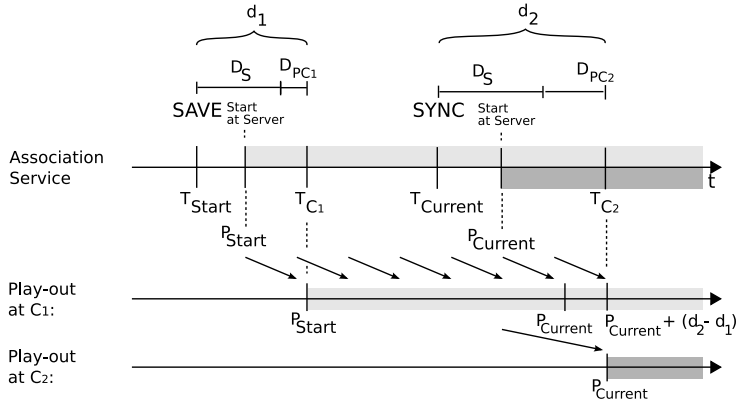


Figure 4.9: Problem of Delay Difference

informed that they should synchronize again. The proxy can also re-synchronize the clients and send a notification containing the new play-time position.

Individual session control is simple, because control operations run completely independent from other clients. Only the state of the new association must be saved.

The advantages of this approach are that repositioning for an individual client can be done easily. In this case, streaming state is also delivered well and client implementations do not need to be changed. However, in case of group repositioning, the other members of the association must be notified about the new state. Another advantage is that no data path manipulation must be done in the association service. An interesting property of independent synchronization is that different streaming servers can be used, e. g. if each client connects to the provider-specific media server.

The disadvantages are that synchronization is not very exact, because the timestamps are independent. However, this can be compensated by the optimizations shown below.

The approach is suitable for scenarios where video is streamed to independent devices and where users are likely to change their position individually. Mostly spontaneous meetings, but also some home networking or learning scenarios may fulfil this concern.

An optimization of the independent mode is the measurement of the latencies between server, proxy and clients. Thus, the calculation of the play-time position can be refined. The delay measurements should be done before the start request is processed. Thus, at the time the synchronization state is saved, a *play-out delay* is calculated which denotes the time difference from the arrival of the request at the association service until the response (and therefore the media data) would arrive at the client. Thus, the play-out delay of the association is calculated from the round-trip delay to the server and the one-way delay to the client as

$$POD_{Assoc} = D_S(t_1) + D_{PC_1}$$

If another client wants to set up a session, the same delay measurements are taken. The play-time is calculated by summing up the start position and the difference of current time and start time.

However, this would not account for varying networking delays to the server and for differences in the networking delays to both clients. Hence, the play-out delay is calculated for the joining client again:

$$POD_{C_2} = D_S(t_2) + D_{PC_2}$$

The calculated play-time PT_c is corrected by subtraction of the saved association play-out delay and the addition of the play-out delay for the joining client to the resulting PT_r:

$$PT_r = PT_c - POD_{Assoc} + POD_{C_2}$$

An example for the calculation of the play-time position is sketched in figure 4.10. Client C_1

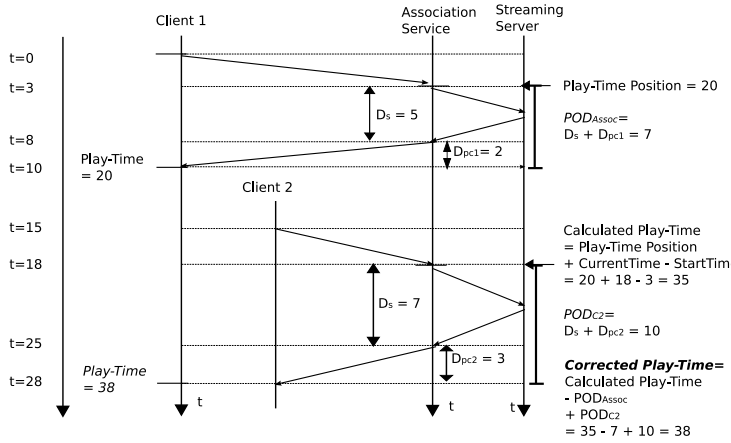


Figure 4.10: Calculating Synchronization Delay

requests to play from position 10. The request arrives at the proxy at time t=3. Suppose the proxy has measured the server-delay as $D_S = 5$ and the one way client-delay as $D_{PC_1} = 2$, then the association play-out delay calculates to $D = 7$.

Client C_2 joins the association with a start-up request arriving at time 18. The server-delay at this time is measured as $D_S = 7$ and the (one way) client-delay as $D_{PC_2} = 3$, which results in a play-out delay of 10. The play-time position calculated from start position, current time and start time accounts to 35, however this must be corrected by subtracting the saved play-out delay of 7 and adding the new play-out delay of 10. Thus, the "real" play-time is 38, which corresponds to the play-time position of C_1 at the time t=28, when the media play-out also starts at C_2 (supposed that clients buffer the same amount of data).

The proposed algorithm can handle variances in the networking delays to the server considering the time scale from the first client's request to the current request. However it does not account for short-scale variances from the time of the delay measurements to the time when media data are actually delivered. Hence, the play-time position may not always be accurate.

Another optimization possibility is to send a virtual delay to the clients synchronizing their play-out to the client with the largest delay in the group. This can best be combined with feedback

about client buffer state. Thus, play-out buffers must be adapted so that the presentation play-out starts at exactly the time directed in the virtual delay. Thus, a more exact synchronization can be achieved. However, this approach requires global clocks and much more complex operations on clients and proxy. Still it remains difficult to synchronize a late-joining client to an association without tighter coupling of the clients' sessions.

If play-out at different devices has to be synchronized exactly during the whole runtime of the presentation, an explicit synchronization protocol must be used, which uses similar concepts, but the synchronization is kept up by adapting buffers and re-synchronizing each time a client buffer gives feedback about buffer space violation. However, for many scenarios it is sufficient to have an exact initial synchronization, because real-time streaming applications can maintain time relations within and among local media streams.

4.5 Summary

In this chapter, the notion of collaborative streaming has been defined with respect to groups and their state. The collaborative state has been defined as a composition of shared and individual state. The shared state consists of attributes that are relevant for the group relation and therefore have to be controlled by the management, whereas individual state attributes can be chosen freely by the members. The concept of an association as a subgroup with members that have the same shared state has been presented. This concept enables the partition of groups. Since this partition should be done in a controlled fashion, policies for access control are proposed. An entity which is responsible for access control checks whether streaming control requests are allowed to be executed according to this policy.

The shared session state of a collaborative streaming group must be enforced by session sharing mechanisms. This means that the sessions of the members of an association have to be synchronized to a common play-time position or a common track set must be chosen. Moreover, for each admitted control request a decision must be made whether the whole association has to change the state or whether the association has to be partitioned. This so-called reaction policy is also used to resolve conflicts within a collaborative group. Clients are notified of changes to the collaborative streaming group that have been initiated by other members. So-called publish-subscribe mechanisms are used for these notifications.

Two mechanisms to enable a common collaborative play-time state have been presented: the reflected and independent synchronization modes. These synchronization modes address the problem to synchronize a late-joining client to a running presentation. Whereas the reflected mode implements a multicast-style synchronization, the independent mode can be used without centralized data transport. Possible optimizations for both modes have been shown.

In the next chapter, the collaborative streaming service is examined from the user and system perspective. The services that implement the presented concepts are described in more detail.

5. Collaborative Streaming Service

In this chapter, a collaborative streaming service is described which applies the concepts presented in the last chapter to the scenarios and user requirements that have been examined in chapter 2.

A collaborative streaming service in general can be thought of as an intermediate system located between the group of clients and a streaming server, which implements a collaborative session control, allows for session sharing and provides for session transfer functionality. Such a system is sketched in figure 5.1.

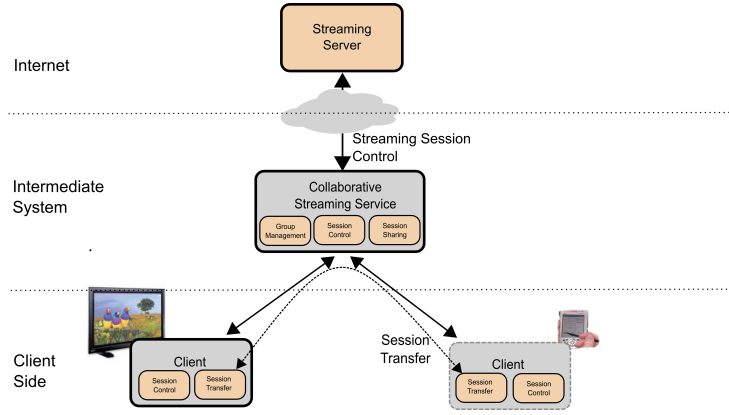


Figure 5.1: Overview of Collaborative Streaming Service

The figure depicts the control flow among the components whereas the data traffic has been omitted for the sake of clarity. Note that the session sharing component of the intermediate system nonetheless influences media data transport. The figure also shows a partition of the client into the two modules of session transfer and session control. This partition reflects the independent user requirements concerning collaborative session initiation (executed by session transfer functions) and control.

In the figure, the collaborative streaming service is shown as a complex service with several sub-services. Altogether, the sub-services must fulfil the functionality shown in the last chapter. In the area of Web Services architectures [166], the composition of several fundamental services to a more complex service has found some attraction [121]. In multimedia systems research, flexible service-based architectures are examined also: A service composition framework provides for

decomposition of complex services along several metrics, which comprises time as primary metric for multimedia services [111]. Additionally, complex services are classified after the number of required services, performance quality, content type, and infrastructure support.

In this chapter, the user-view of the collaborative service is described first to define the scope of the functionality of each of the sub-services. Additionally, the inter-working of the sub-services to implement the required functionality is shown in sections 5.2.1 and 5.2.2. Afterwards, the individual sub-services are described in more detail to lay the foundation for the exact architectural representation with existing Internet protocols.

5.1 User View of Collaborative Streaming Service

The user view of the collaborative streaming service can be described according to a decomposition of the complex service according to time metrics. This time metrics decomposition has been proposed in [111] for complex multimedia services in general. Hence, services can either be *successive*, *concurrent* or *hybrid* regarding composition along the time metric. Successive services are executed exactly one at a certain time to deliver a complex service. Concurrent services run in parallel, whereas a composition of hybrid services uses the results of concurrent services as input to another service. The authors mention instant messaging as successive, video conferencing as concurrent and interactive HDTV as a hybrid service composition.

In the following, we compose a complex collaborative streaming service from streaming functionality on the one hand and group conferencing on the other hand. We enhance this composition with a view on functions that are called by users to identify functional dependencies between the individual services.

For an individual client without any group interaction, streaming session control offers the functions of initiation, update (i. e. pausing, changing the track set, etc.), and of completion of the streaming service as shown in figure 5.2. After the initiation function has been successfully completed, the streaming service runs in parallel to the streaming control service until the completion function terminates the streaming service. During the runtime of the streaming session, several operations which update the streaming service may be called. Thus, streaming control and data services can be seen as concurrent services.

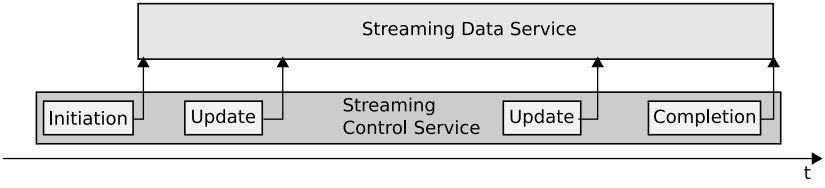


Figure 5.2: Streaming Services for Single Client

A collaborative streaming session with group interaction should have a similar user interface like a standard streaming service, however the described functions of initiation and update must be collaborative and therefore include a group management, as shown in section 4.2. A certain form of group management is also known in group conferencing, which comprises of a group

communication service like a common discussion channel and a control service which manages information on conferencing state (see figure 5.3).

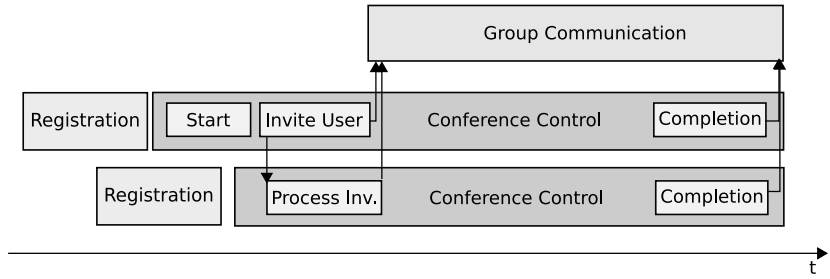


Figure 5.3: Conferencing Service Functions

Such conferencing services require the initial registration of participants such that calls can be routed. This registration is necessary at the start of the client application and each time the user moves to a different device or access network. After an invitation of another client, the group communication service itself is started. Some conference control services allow to request information or change parameters of the communication service also. The control service also provides for functionality for group communication termination.

If the functionality of streaming and conferencing service for a collaborative streaming service is composed, additional functionality has to be provided to allow session sharing and collaborative session control. At the start of the collaborative service, a registration similar to the conferencing service registration must be done. The initiation of the streaming service and the group member invitation is subsumed into one *initiation* transaction. The term *transaction* is used here because the function call at the user normally includes processing at several sub-services of a collaborative streaming service. There are several possibilities for such a transaction, which will be presented in section 5.1.1. All of these have in common that they change the member set of a collaborative streaming group with the result that the number of members is equal to or greater than it has been before execution.

This transaction starts the streaming control and subsequently the streaming service for the joining member itself. Besides that, the group communication service is started if desired and runs in parallel. To offer session sharing the joining member must be synchronized to the group, which is done by the so-called *association service*. This service implements session sharing and synchronization functionality as introduced in section 4.4. It maintains associations and saves or retrieves synchronization state. Control operations of the streaming control service require also an update of group state in a collaborative service. Since this is a complex functionality, it is presented as a so-called *update* transaction to the user. An update transaction is not called for any change in group membership, instead, initiation or *completion* transactions are used. Several choices exist for completion transactions: On user request, it stops the whole collaborative streaming service, the group management and session sharing, or the streaming service only.

An exemplary use of the collaborative streaming service for a whole group is depicted in 5.4.

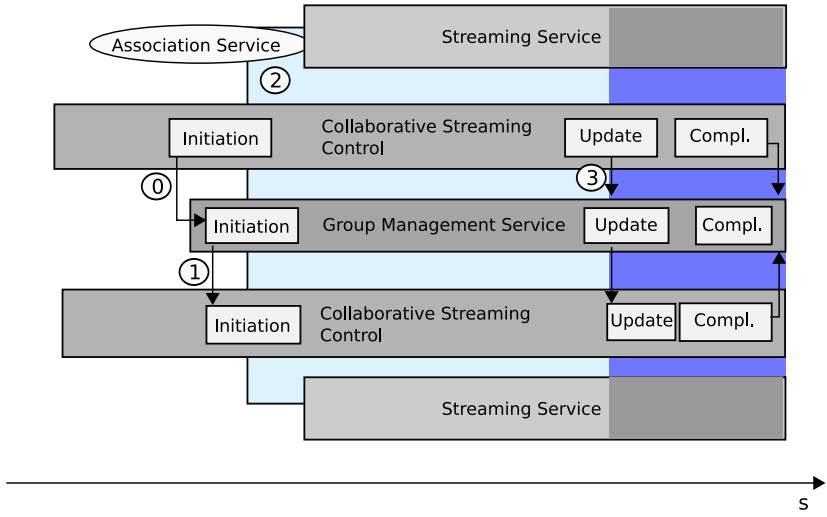


Figure 5.4: Collaborative Streaming Service Schedule

In this case, two clients run their collaborative streaming control and register to the collaborative streaming service. The initiation transaction of this service is invoked for the whole group by one client. In turn, the group management service is started for this group (1). It offers session transfer functionality, as depicted by the invocation of the initiation transaction at the client at the bottom side of the figure (2). In order to set the collaborative streaming session state for the whole group, the association service is started as a helper service (3). It is shown as a background service in a lighter color during the whole collaborative streaming session, because it normally is not present at the users' view. The association service influences the streaming service, which consists of streaming control and data service. The exact functionality of the association service is described in section 5.3.2. The picture also demonstrates that in this case, the update transaction executes a state change for the whole association, depicted by the change to darker color (4). Besides, the state change is also conducted at the streaming service for both clients. This behavior is subject to the policies negotiated at group initiation, as it is described in section 4.2. Parallel to the streaming service, a group communication service can be started to enable discussions. This group communication has been left out of the figure for the sake of clarity.

Since the group management and the streaming control service must inter-operate with the association service at least to exchange data, the borders of services are not consistent with the borders of usual Internet services. This means that intermediate components have to be implemented in the collaborative streaming architecture. Additionally, these components must inter-operate to provide for the composed collaborative streaming service.

A transaction requires interaction with several service components, which makes the successful completion of it dependent on the successful manipulation of the service entities. For instance the initiation transaction does not complete successfully without a successful start of a streaming service. Dependency graphs of the transactions are shown in section 5.2.1.

5.1.1 Initiation Transaction

With the execution of the initiation transaction, users start the collaborative service on their client system. The initiation transaction may be run for one client or for a group as a whole. If no group exists, the initial state is set up, otherwise the client is added to the group. If a group communication service is desired it can be started at the time of group initiation or added at a later time.

The initiation transaction is one of those types which were already shown as use cases in section 2.4:

Start *Start* and *StartGroup* are the transactions used for initiation of a new group. While Start is intended for groups with only one starting user, StartGroup also sends invitations to other participants given in a parameter list. Additionally, the communication service can be started for the StartGroup transaction. The interaction with Group Management is required for both of these initiation transaction types, because the initiator of the group will set up the management policy. The group management also registers the group at the association service to initiate session sharing. Both transactions are appropriate in all scenarios. If a service provider only offers unsynchronized Copy and Pull transactions, the Start transactions can be simple: No associations are needed and no shared session state must be saved. The group management functionality can then be minimized to provide information for location and configuration.

Push *Copy* and *Move* are the two possibilities to transfer the streaming session to another participant. The Copy transaction is used by a member of the group to add a new member to it. It is required for all scenarios. In most scenarios, a synchronized copy (*CopySync*) is used, such that the client shares the same state as the other association members. If an unsynchronized Copy (*CopyUnsync*) is required, as might be the case for spontaneous meeting scenarios, a new association is opened for the joining user. Users joining later can use synchronized or unsynchronized transactions then. The Move transactions, either with or without moving of control, are important in home networks, because they are particularly relevant for scenarios where one human user may control several devices. If the session control is moved to the joining client, this client replaces the old client in the group, otherwise the old client must keep session control and media data must be sent to the joining client. In other scenarios than home networks, move usually does not have to be implemented.

Pull With a pull, a new member joins the group. The streaming session is transfered from an already existing participant to the initiator of the transaction. Similar to the Push transactions, there are also two types of the Pull transaction: *Copy*, which can be done in a synchronized and in an unsynchronized form, and *Move*, where both control and data are moved to the requester.

The cases of *Start*, *Push/Copy*, and *Pull/Copy* are shown in figure 5.5. At the start of the group by the first client, the management is initiated for this group. The synchronization state is saved by the association service. Further push or pull transactions add members to the group. Additionally, the association service maintains synchronization for the joining member, if synchronized push and pull is desired (as assumed in this figure).

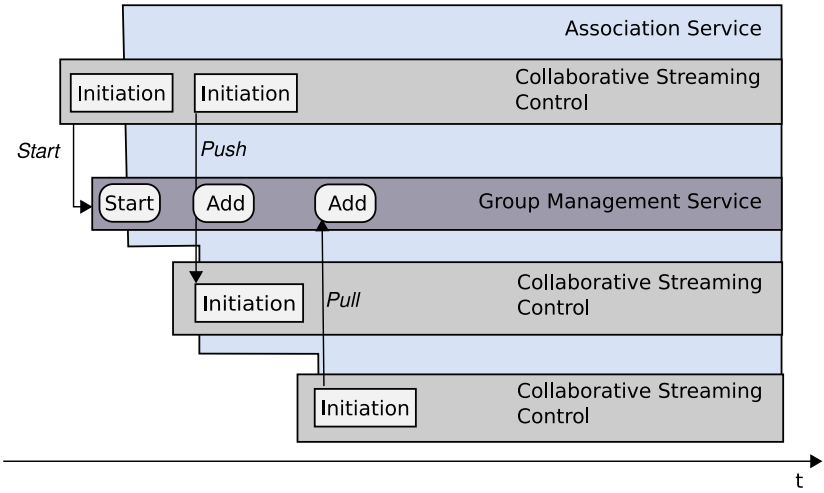


Figure 5.5: Different Initiation Transactions for Clients

The streaming service (not shown in the figure) can be started if group management and association service have successfully completed the required operations. An initiation transaction is considered as completed if the streaming service has started successfully.

5.1.2 Update Transaction

The session control functions of standard streaming services can be seen as update functions of streaming session state. In collaborative sessions, users also execute streaming session control functions and thus call an update transaction of collaborative streaming. The update transaction is completely independent from the initiation transaction. Different from single-user streaming, an update may not only influence the own streaming service but also the service which is provided to other group members, according to the rules specified for a certain scenario.

Not every group member is allowed to execute update transactions. This is dependent on the update control function and the scenario. The set of possible update transactions depends on the set of control functions which are provided by the streaming service. Some plausible control functions are related to time, like pause, jump, or search (fast forward or backward), or related to the track set, like functions to add or remove tracks. Additionally, functions that are specific to collaborative streaming can be provided, like explicitly open a subgroup or synchronize to a certain association.

In table 5.1 exemplary access permissions for the different scenarios are shown. Jumping and searching have been subsumed under the notion of *repositioning*. In home environments, pausing and synchronizing to other's state can be allowed for all clients, whereas jumping is appropriate for clients controlling a video device. In learning environments, most updates are allowed for an administrator only, since this administrator wants to control the process of the presentation. However, pausing to ask questions often makes sense. In this scenario also a specific request to open a new subgroup and put members into this subgroup is appropriate. Changing the track set

Control Request	Home Scenario	Learning Environment	Spontaneous Meeting
Pause	for All Clients	for All Clients	for All Clients
Repositioning	for Video Clients	for Administrators	for All Clients
Open Subgroup	–	for Administrators	–
Synchronize	for All Clients	for Administrators	for All Clients
Change Track Set	–	for Administrators	–

Table 5.1: Access Control for Updates in Scenarios

is mostly an individual decision of clients, e. g. , according to their device capabilities. Thus, the track set is not under access control except in learning scenarios, where supervisors may switch to a different media type. Spontaneous meetings allow each user to perform update operations whenever it is desired.

As already described in section 4.2.1, access permissions can hardly be managed based on single users' addresses. Users are classified according to roles. Therefore, different roles are defined for each scenario, which contain a set of permissions and a member set. In general, it is difficult to find universal role definitions for a certain scenario. Since members can assume different roles, a simple approach maps roles one-by-one to permissions. Still, an explicit administrator role is required to perform management operations.

For processing of the update transaction with time, the single control function is less important. Moreover, the reaction of the group state has consequences on the processing. Two kinds of reactions can occur on a typical session control action like pause or jump: either changing the state of the association or changing the association itself (which includes opening a new one, if none exists that has the desired state). Session control actions that are more specific to collaborative streaming like synchronizing to another member's state already imply the reaction of joining the desired association.

The update transaction processing with time is shown in figure 5.6. We assume that access control

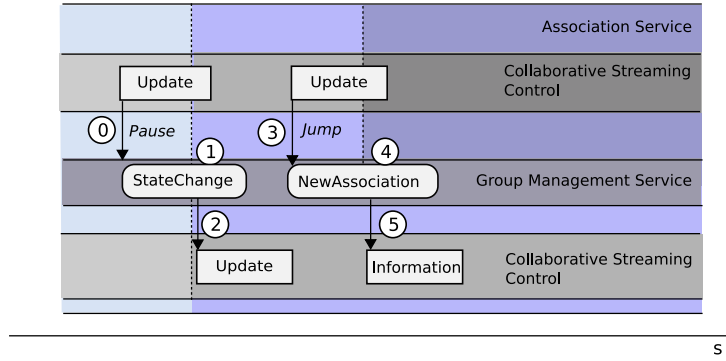


Figure 5.6: Update Transactions Processing

has been successful in each case. First, one client sends a *Pause* (1). In this case, the reaction

that the whole group will pause the stream is assumed. Thus, the operation to change state is invoked at the association service. In the figure, the association changes state for the whole group, reflected by the change to darker color (2). All streaming services of the clients must be notified and updated with the new (resume) position (3). In the second case shown in the figure the client wants to *jump* to a new position (4), but the group stays at the viewing position. Hence, the reaction to the jump request of the client is to open a new association at the association service and request a state change only for the streaming service for this client (5). In the association service, two associations for this group exist (see the different shades of gray in the figure). A notification of other clients is not essential, but could be done for information (6).

Control Request	Home Scenarios	Learning Environments	Spontaneous Meetings
Pause	Change State	Change State	New Association
Repositioning	New Association + notify	Change State	New Association
Change Track Set	–	Change State	–
Open Subgroup	–	New Association with defined Members	–
Synchronize	Join Association	Join Association	Join Association

Table 5.2: Typical Reaction to Update in Scenarios

Possible reactions to update control requests for each scenario are shown in table 5.2. The requests to open a subgroup and to synchronize to a certain association already involve their reactions. In home scenarios, a distinction is made between pausing and re-positioning, whereas in learning scenarios the state is always changed. In spontaneous meeting scenarios an appropriate reaction is to open new associations for every control request. If role definitions are more elaborate than proposed before, a distinction of reactions according to user roles can be expedient. However, this case is left for future considerations.

Like the access control permissions, these reactions are dependent on the scenario and on personal requirements. If a session attribute is not considered as shared, no reaction has to be defined.

5.2 Functional Domain

To implement the mentioned transactions, the collaborative streaming service has to provide for certain functions on system side, like the addition or registration of members in the group management or association service, respectively. An overview of the individual collaborative streaming sub-services is shown in figure 5.7. The collaborative streaming service is divided into user and system part. The system offers initiation and update transactions that can directly be accessed by session transfer and control functions on the user side. The system itself implements group management and session sharing functionality to manage a consistent group state. The single functions have to exchange information, for example results of access control.

In this section, mainly the system side of the collaborative streaming service is examined. We provide for functional descriptions of the individual services which are protocol-independent. This means that implementors could choose their own session transfer and control protocols. An architecture of the collaborative streaming service with IETF standard protocols will be shown in chapter 7.

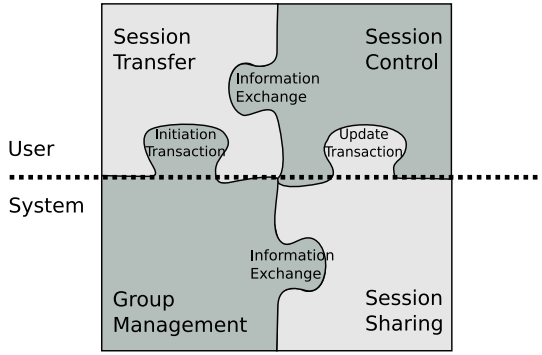


Figure 5.7: Collaborative Streaming Service Parts

5.2.1 Initiation Transaction Processing

In the following, we describe how the initiation transactions can be implemented on the system side. If the individual services directly provide the required functions as described in section 5.1.1, the function graph shown in figure 5.8 emerges:

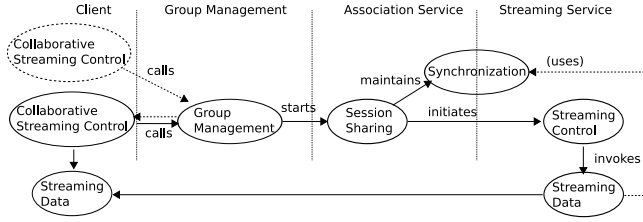


Figure 5.8: Direct Function Graph of Initiation Transactions

All clients have a collaborative streaming control functionality, which contains session transfer and session control operations. They invoke the initiation transactions on the group management. Depending on the kind of initiation transaction, the group management can be called by other clients (shown as an ellipse with dashed line). It changes the set of clients and starts the session sharing at the association service on behalf of the added client(s). The session sharing calls synchronization functionality and initiates the streaming control service, which finally invokes media data transfer.

This distribution of functionality natively supports initiation transactions without requirement of any further client interaction. Though this is an obvious advantage, the approach also has its disadvantages: The streaming session state has to be provided to clients so that they can control media data transport (e. g. open required ports). In this case, either the group management has to implement session control functions or the session sharing has to open a control channel to the client to deliver the necessary control information. This processing requires the development of a

number of additional signaling protocol methods. It makes the use of existing components for clients and intermediate systems difficult (if not impossible).

Hence, we propose to separate session transfer and session control functionality in the clients. The client itself starts the streaming session, as shown in figure 5.9.

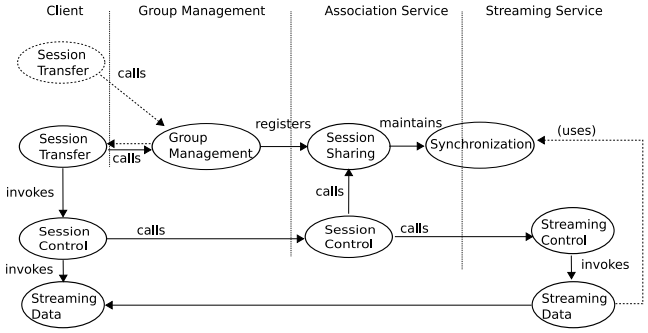


Figure 5.9: Function Graph for Initiation Transactions

At the client side, the initiation transaction is transformed into a session transfer operation, which calls the corresponding functionality of the group management. The group management registers the modified group information to the session sharing, which is implemented by the association service. The association service must also implement session control functionality, which calls the session sharing. Since the session control otherwise works as a server relating to the client, existing client-server implementations for session control can be re-used in that architecture. In turn, the session sharing maintains necessary synchronization information. Eventually, the streaming service is started and uses the information from the synchronization service or functions of the service itself, dependent on the synchronization mode.

In table 5.3 the required service functions for each of the initiation transactions are given.

Transaction	Group Management	Session Sharing	Streaming Control Service
Start	Initiate Group	Open Association	Setup and Start
Start Group	Initiate Group, Add Members	Open Association, for each Member: Join Association	for each Member: Setup and Start
Push / Pull, synchronized	Add Member	Join Association	Setup and Start at Synchronization Point
Push / Pull, unsynchronized	Add Member	(Open Association)	Setup and Start
Move	Replace Member	Replace Member in Association	Reset Control and Destination
Move, keep Control	-	-	Reset Destination

Table 5.3: Functionality of Initiation Services

We have assumed that the initiation services are accepted by the relevant users and that each service will pass with success. The push and pull transactions are distinguished by their use of synchronization and copy or move only. Synchronized transactions require calls to session sharing operations, whereas unsynchronized transactions can leave out session sharing if no other clients are expected to synchronize later. Otherwise, a new association is opened. Move services can be implemented by changing the destination. The old client must be replaced by the new one in the group if the new one is provided with control functionality.

5.2.2 Update Transaction Processing

An update transaction is initiated by a session control function at a client. Similar to the initiation transactions, update processing can be directly constructed from the service decomposition already shown in section 5.1.2. Clients direct their session control request to the group management as shown in figure 5.10.

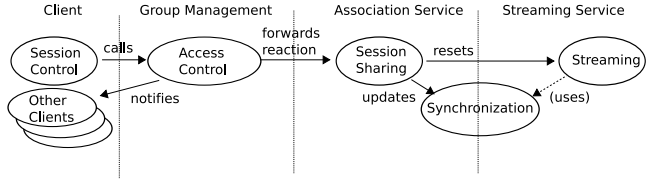


Figure 5.10: Direct Function Graph of Update Transactions

The group management performs access control and forwards the reaction to the session sharing service, which in turn updates the synchronization service and resets the streaming service. Again, the group management service would have to implement session control functionality besides access control so as to pass the control operation to the session sharing service.

As proposed in the last section, the association service implements session control functionality. Thus, it is appropriate that control operations are directly forwarded to the session control of the association service. This module calls the access control operation at the group management, as shown in the function graph of figure 5.11.

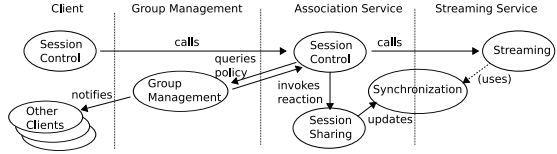


Figure 5.11: Function Graph for Update Transactions

The group management keeps policy information, therefore access control requests must be directed to it. In case of successful access control, it returns a reaction operation to the session control. This reaction determines the operation of the session sharing service, with an update

Reaction	Group Management	Session Sharing	Streaming Control Service
Change State	for each Member: Notification	Change State	Control Operation
Change Association	Optionally: Notification	Open Association or Synchronize to Association	Start at New Position

Table 5.4: Functionality of Update Transactions

of the synchronization service. The group management sends a notification to the other clients, especially for a state change for the whole group.

If reaction times of control requests are essential, the group management can convey the policy information to the association service at the time of group initiation. However, the effort for communication and storage would be higher.

In table 5.4 the required functions for each of these services are given. Each control operation is processed in a similar fashion. Hence, we only distinguish among different reaction policies for each control function.

Changing the state means that each member must be notified about the new state, and the new state must be saved for future synchronizations to this association. The control operation is forwarded to the streaming control service. If the user has to change the association the session sharing may either open a new association or synchronize to an existing one. In either case, the streaming service for the user is reset to a new position.

5.3 Collaborative Streaming Service

In the following subsections, functional descriptions of the sub-services already mentioned are given. The basic style of the functional description is derived from the *service profile* of OWL-S [97]. OWL-S is a set of markup language concepts and provides for a description of properties and capabilities of web services to enable automation of all web services tasks, like discovery, composition, inter-operation, and others. In the service profile of OWL-S, the service is described in a way that clients may decide if this service fulfills their demands. The service profile thus describes a kind of public interface, concerning the information that is exchanged between client and service (the *input* and *output* parameters) and the state before the service has been invoked (*preconditions*) and after the service has been finished (*effects*). This description is also used in the *process model*, which describes how the service works in greater detail. For the sake of readability, we use textual description for the profiles of the decomposed services [34] and leave the actual descriptions using the Resource Description Framework (RDF) [27] for future work.

5.3.1 Streaming Service

The streaming service of a media server is controlled by session control operations which are executed by a client. The service can be divided into two sub-services, the streaming control and the streaming data service. The latter is responsible for the effective media data transport from server to client and is invoked and controlled by the streaming control service. A real-time *streaming data* service provides for the functions of media stream packetization and scheduling

so that the transmission of the data portion takes on average the same time as presentation of the data. This service is assumed to exist at certain media server locations on the global Internet. As such, it is less relevant for the functional description of the collaborative service, however there may be intermediate services that change the data provided by the streaming server. An example is a reflector service for the reflected synchronization mode (cf. to section 4.4.2.2) that actively contributes to the implementation of the streaming service. Such a reflector sets up a session with a defined start position and by definition copies incoming packets of this session to all outgoing destinations. Further it implements the functionality of adding to and removing clients from the set of outgoing destinations of a specified session. A different example may be a transcoding service that offers certain data formats optimized for mobile users [162], [29].

The *streaming control* service offers a set of functions to control a real-time streaming data service as listed in table 5.5. These functions constitute the session control on client side. Note that the so-called aggregate control of the tracks of a streaming session is assumed to be given by the streaming control service. It may also implement non-aggregate control, i. e. open a new session for each track, but we leave this case for further consideration.

Method	Input	Output	Preconditions	Effects
Setup	Presentation URI, Track List, Transport	Session Identifier	Presentation / Tracks exist	Initiate Data Service
Start	Session ID, (Playtime)	Timing Information	(Playtime within Range)	Start Data Service
Pause	Session ID	Resume Position		Pause Data Service, Keep Information
Resume	Session ID, (Resume Position)	Timing Information	Resume Position within Range	Reset Data Service
Jump	Session ID, Playtime	Timing Information	Playtime within Range	Reset Data Service
Search	Session ID, Scale factor	Timing Information	Factor Supported	Reset Data Service
Finish	Session ID			Stop Data Service
Change Transport	Session ID, New transport			Reset Transport at Data Service
Add Track	Session ID, New track, Transport		Track Exists	Add Track at Data Service
Remove Track	Session ID, Track		Track Set Up	Stop Track at Data Service

Table 5.5: Streaming Control Service Description

The setup function requires an URI of a presentation together with the list of desired tracks. Additionally, transport information like destination address and ports are committed. A precondition for successful session setup is that the presentation and its tracks must exist. The transport parameters must be accessible, but in case of unreliable transport this condition is not verified. It returns a session identifier, which has to be used in all following control operations. This enables moving of the control to a different client device. The timing information returned by the start operation is necessary to perform inter-stream synchronization locally at the client system. Pause gives back the play-time position for resuming the presentation later. Other control requests are jumping to a certain position and fast forward or rewind functionality, which is subsumed under the notion of searching here. Transport parameters can be changed to reset the transport to a different destination. Finally, tracks can be added to or removed from the streaming

service. Preconditions for the single operations require that the input parameters must exist or be within a certain value range. If any given precondition is not fulfilled, the service responds with an error message. Errors concerning the streaming session setup lead to failure of the service, whereas errors concerning session control requests during play-out do not necessarily cause session tear-down.

This set of functions reflects which standard streaming control functions are valuable to a collaborative streaming service. Available real-time streaming standards or implementations may differ in some aspects, which will be described in section 6.1 for the widespread real-time streaming standard RTSP and its implementations.

5.3.2 Association Service

To turn the single-user streaming control service presented in the last section into a collaborative streaming service, the association service is required. It has to implement a slightly modified session control, which relays session control requests of the client to the streaming control of the server. Additionally it calls session sharing operations and provides for access control by querying the group management's policy.

Moreover, a session sharing and a synchronization service are required, which are both presented in the forthcoming subsections.

5.3.2.1 Session Sharing Service

Another aspect that distinguishes collaborative from single-user streaming is the synchronized play-out of tracks for those group members who share an association. Hence, associations must be maintained by the session sharing part of the association service.

Method	Input	Output	Preconditions	Effects
Register Group	Group Member(s)	Group Identifier		Group Added
Set SyncMode	Synchronization Mode		Mode Exists	Association Members are Synchronized by Mode
Open Association	Association Member(s), Session State	Association Identifier		Association Added
Remove Association	Association ID		Group / Association Exists	Remove State
Join Association	Client, Association ID	Session State Information	Group / Association Exists	Synchronize Client to Association Session State
Leave Association	Client, Association ID		Group / Association Exists	Remove Client from Association Member List
Change State	Association ID, State Variable, Value		Value Allowed	Changes the State

Table 5.6: Association Service Description

The functions described in table 5.6 provide for the interface of the session sharing service. Most of the operations are accessed by session control. Exceptions are the group registration, which is done by the group management, and setting the synchronization mode, which is done by an administrator globally or by the group management for each group. We distinguish between

group registration and opening of associations. Group registration can be done at an earlier time to allow members to join before actual session start. The association is connected with a certain session state. As soon as the session control has defined this session state, the association is actually opened. The join, leave, and remove functions can also be executed without this session state, which means that they operate on the group level. Changing the state can only be done if a certain session state has been initialized.

5.3.2.2 Synchronization Service

The association service is responsible for saving the synchronization state of associations. This state is used as an informational means for client applications as well as a resource for synchronization calculation in some synchronization modes.

The synchronization functionality itself is executed each time a client joins an association or the session state is changed. It has mainly two operations: first, the session state must be saved at the start of an association or if the session state within the association is changed. Second, the current session state, which may be calculated from the saved session state, must be retrieved if a member joins the association. These procedures must be done according to the selected synchronization mode. The collaborative streaming architecture provides for an implementation of these modes, which will be described in greater detail in section 7.4.2.3.

For track synchronization, the track list relevant for an association is returned as an output. If a user has been in a different association before, the set of selected tracks must be changed during runtime, which will call the add or remove functionality of the streaming control service. The processing of transport parameter synchronization is very similar, the function to change transport parameters is called and the changed transport information will be returned.

5.3.3 Group Management

The ideas of the group management have been described in section 4.2. Most of its functions are session transfer functions, which can be accessed by the clients' initiation transactions.

Method	Input	Output	Preconditions	Effects
Initiate Group	Presentation URI, Policy	Group URI		Creates Group
Add Member	Group URI, Client	Notification	Group Exists	Adds Member with Role
Remove Member	Client	Notification	Client in Member Set	Removes Client
Replace Member	Old + New Client	Notification	Old Client in Member Set	Replaces Old with New Client
Create Subgroup	Client Address	Subgroup URI	Group Exists	Subgroup Added
Access Control	Client, Control Operation	Reaction Notification	Policy Exists	Execute Reaction
Close Group	Group URI	Notification	Group Exists	Removes all Group State

Table 5.7: Group Management Service Description

The necessary functionality for maintenance of groups is shown in table 5.7. At time of initiation, the streaming presentation URI and a management policy must be assigned to the group. The

functionality of add and remove is straightforward. A notification about the new membership state is distributed to all subscribers of the group state. The replace functionality is specifically used to realize the Move initiation transaction, thus state can easily be taken to a new member. The group management also offers an explicit function to create a new subgroup, which is done in accordance with the reaction to open a new association. Since this subgroup is assigned an own URI, the membership operations shown above can operate on subgroups also. However, a subgroup cannot exist without a main group. Hence, if a group is closed, all subgroups will also be removed. Group communication services can also be configured to exist for subgroups so that members of a subgroup can talk to each other exclusively.

After a group has been registered at the session sharing service, membership state changes must be forwarded to the association service to prevent that unregistered members access the collaborative service.

The access control functionality is not a public functionality, but invoked by a session transfer or session control operation. It validates a control operation as shown in the last chapter, and executes group state changes according to the reactions defined in the policy.

5.3.4 Group Communication Service

The group communication service offers a communication channel between a set of participants. It can be implemented on a central entity like a conferencing server [7] or distributed on a peer-to-peer base [92],[149].

The functions that have to be provided by this service are the setup and tear-down of a common channel and functions for members to join and leave this channel. Additionally, transport or media parameters are set and negotiated. Media data of all participants are combined by entities called mixers, which can be implemented either centralized, hierarchical or distributed [132]. Optionally, talks among a subset of the group members may be enabled by providing extra channels.

We assume that a group communication service can be provided by an existing media conferencing application and do not go into further details of this service.

5.4 Configuration and Discovery

A collaborative streaming service consists of several services, which may be implemented using applications with different address destinations. It is important to configure such a system in an automatic fashion, because non-expert users are not able to manage configuration in a probably changing environment. Particularly in mobile environments, users are hardly capable to type in long presentation identifiers because of restricted input devices.

In this section, concepts to locate the necessary building blocks of a collaborative streaming architecture as well as users and presentations are presented. Specific attention is paid to the location of an existing collaborative streaming session, i. e. a presentation that is streamed to a certain user or group of users. The description of services and resources is important for their location. We have already provided for a coarse description of the service components. In future work, even fully automated discovery may be supported by applying OWL-S service profile descriptions [97]. For presentations, meta-data descriptions exist [108], though they are not widely implemented today. Addresses of users are mostly kept in address books. Unknown

addresses could be located by applying a search engine, however address records of users are often incomplete.

A location system that seamlessly fits into the collaborative application and locates sub-services on-demand would be very complex in our case, because several system components depend on each other (confer the foregoing sections). Therefore, we leave this for future work and only describe the necessary mechanisms to locate services on user side.

Method	Collaborative Service	Streaming Service	User Address	Session Address
Start	x	x	-	-
StartGroup	x	x	x	-
Push	x	-	x	-
Pull	x	-	(x)	x

Table 5.8: Location of Services and Addresses

The destinations that have to be located in a collaborative streaming session are shown in table 5.8 for initiation transaction classes. The initiators of all transactions must locate the collaborative streaming service, consisting of the mentioned association and group management sub-services. At time of collaborative session start, a user searches for a certain presentation, which is offered by a specific streaming service. Addresses of other users only have to be available if a client wants to start the session for a whole group of users. The session address is created as a result of a Start transaction and thus cannot be located. Push and Pull transactions do not have to locate a streaming service, because the presentation information is known and can be distributed. However, the initiator of a push transaction must be aware of the target user's address or find it out. A user who wants to join the session with a pull transaction must locate the session address, i. e. the group identifier. It may also have to locate an address of a participant of a session first, if required by the address location mechanism.

5.4.1 Service Location

Users must be able to locate a registration service as an entry point to the collaborative system, quite similar to existing conferencing systems where users are required to register themselves, mainly for application routing purposes. Additionally, an intermediate system to provide for the collaborative streaming service as described above must be located and possibly provided for the user as a configuration means or added to configuration records.

Another question to answer is if a central entity is to be used to register and discover services. The alternative would be that services themselves answer service requests. The first provides a better distinction from service implementation to service discovery, because services just need to register to that entity. The second alternative saves some overhead for service registration, and may deliver information about the service in time. However, services must implement all location functionality themselves.

5.4.2 User Location

If a session is to be pushed to a user, an address to direct the invitation to must be found. Even if the name of the user is known, some parts of the address may be missing. It would be possible to

put names of known persons into an address register, but probably not all possible collaborative streaming partners are known by address. Thus, it should be possible to locate users in a certain network domain (or nearby location). Technically, user device location in the proximity of a client can be done by wireless technologies like infrared data communications [86], Bluetooth [148] or specific sensors [87]. LDAP user directories could be used to query user information in a certain domain, similar to Internet Locator Service (ILS), which was offered by Microsoft for the NetMeeting application but has been discontinued [103].

In some collaborative streaming scenarios, sending a search request via a certain wireless technology is adequate. A visitor to a home network may quickly exchange address information via infrared or Bluetooth. In contrast to this, registration information can be utilized in scenarios like learning environments. In any case, filter attributes like the name of a user are helpful for human beings to issue meaningful search requests.

The requirement for user address or status detection always suggests the question about privacy. A user who is ready to participate in collaborative streaming in general may nonetheless be busy at times or does not want to be invited by a particular person. Thus, such a user that somehow expresses his / her wish not to be discovered must not be found by means of the user location mechanism. We assume that location services based on Bluetooth or sensor devices apply privacy mechanisms. If registration information of the collaborative streaming service is utilized, the registrations must be tagged if a user is busy or wants to stay invisible. Additionally, a list of friends can be given to restrict the scope of users who may query the own address. Additionally, queries to the registration database must be authenticated.

We state that with those mechanisms privacy is not enabled in a completely secure fashion. Malicious users could still draw address information out of protocol messages unless these are not encrypted. However, in normal operation a user has to be taken into the friends list by the user to be located, such that this user has at least a certain control.

Generally, a number of mechanisms can be applied or developed for user location in a collaborative architecture. However, they are out of scope of this thesis and are considered for further work only. We assume for this thesis that registration information can be queried.

5.4.3 Collaborative Session Location

Users who want to pull a presentation must have a way to locate the presentation. Locating the presentation has two issues: First, the URL of the presentation on a media server must be found, second, the collaborative session must be located to find out the group state. The latter can be considered as more important for group collaboration, the former for media streaming. A request for a specific presentation together with a reference to a single group member can be mapped to the group identification. Thus, we use the term “collaborative session location” to refer to both group and media URI location.

The following requirements can be defined for a collaborative session location service: Collaborative sessions that are restricted to be joined by a certain user group should only be found by users that have the corresponding privileges. The result of the location should include both group identification and media URI, such that the user can easily join the group and stream the presentation.

Searching for collaborative sessions can be designed using a specific search method, possibly by querying a known user for the presentation and group identification. Another possibility is the

registration of the collaborative streaming session by the group management, similar to client registration itself.

5.5 Summary

A collaborative streaming service is a complex service subsuming several sub-services, which together provide for session transfer, session control, group management, and session sharing. Users directly influence session transfer and session control by executing initiation and update transactions, respectively. On system side, these transactions invoke functions of the group management and session sharing sub-services. Moreover, necessary configuration and discovery functionality has been examined, particularly for the identification of a collaborative streaming session to pull this session from another member.

We have shown that a one-by-one mapping of user-centric transactions into system functionality allows native support of collaborative streaming functionality of the intermediate system. However, it would also require to design a new streaming system without the ability to re-use existing components. A different way to go is to use already existing multimedia signaling protocols as a communication means between clients and intermediate system. The group management and session sharing services are separate sub-services that have to exchange data for registration and control purposes only. The mentioned services definition has to be translated into a signaling architecture. A representation of such an architecture will be given in chapter 7, after the presentation of the relevant Internet signaling protocols in the next chapter.

6. Protocols for Multimedia Sessions and Services

For several years, the Internet Engineering Task Force (IETF) [55] has been defining protocols for signaling of multimedia sessions and for multimedia transport. The *Real Time Streaming Protocol (RTSP)* [145] for signaling and control of real-time streaming presentations and the *Session Initiation Protocol (SIP)* [135] for the management of two-way or multi-party calls are the most important signaling protocols for collaborative streaming with Internet protocols. In both cases, media data are delivered by using the *Real-Time Transport Protocol (RTP)* [144]. Additionally, helper protocols for the description of sessions and presentations like the *Session Description Protocol (SDP)* [52] and for location of services like the *Service Location Protocol (SLP)* [50] are also used in the *costream* architecture and hence described in this chapter.

All IETF multimedia signaling protocols are text-based protocols, which enables readability by humans and allows easy writing of parsers. Request-response based protocols exchange messages to call a specific method in the server or client machine. A response is given to inform the originator of the request message of success or failure of the method or to give further information, for example a redirection to a more specific location of a server. Messages consist of a number of headers which are separated by an end-of-line character. Two end-of-lines together means the end of the message. A message may be accompanied by a body for some methods, for example to deliver a description. An optional body is indicated by the header fields `Content-Type` and `Content-Length`. All those characteristics are similar to the well-known *Hypertext Transfer Protocol (HTTP)* [39].

6.1 Streaming Presentations with RTSP

In this section, we present the protocol characteristics of currently available real-time streaming systems in the Internet. By choice of the term “real-time” we exclude Internet streaming as HTTP downloading of a file with starting the movie when a certain amount of packets was received, sometimes referred to as *progressive streaming*. Though this kind of media retrieval is quite widespread, we argue that in environments with varying resources real-time streaming enables a smoother resource consumption, because only the amount of data that is required for the presentation at a certain point in time is delivered.

Common real-time streaming systems use the Real Time Streaming Protocol (RTSP) for signaling [145]. Media data transport is done with the Real-Time Transport Protocol (RTP, confer section 6.1.4) by default. Unlike HTTP, RTSP is a stateful protocol and allows sending of requests by servers also. Signaling by RTSP enables session control with pausing and repositioning

functionality. Fast forward is also an option in RTSP by applying a scale factor to playing, but most implementations do not provide for this. For the human user, the interface to a client resembles a usual media player or VCR.

Use cases mentioned in the standard are unicast streaming of recorded presentations and live streams delivered by unicast or multicast transport. Another case is the invitation of a media server into a conference, however conference setup and control must be done by separate protocols.

Streaming systems may offer several tracks for one presentation. Nowadays, media files are mostly edited with a static and minimal set of tracks (e. g. one video, one audio track). Thus, clients mostly request whole files, setting up the complete set of tracks available. In future, users will probably ask for presentations tailored to their needs, each one requesting different language audio tracks and different media formats or resolutions. Streaming servers thus may offer several different tracks for one media type in conjunction with a selection tool for clients. Changing the quality of a media data stream at the time of transport as an alternative has been a research topic for several years with layered coding [99],[43] or transcoding [162] approaches, but no commercially available implementation incorporates these. Object-oriented codecs like MPEG-4 [68] provide for inherent scaling and thus may enable such features more easily in future.

6.1.1 Protocol Characteristics

A stateful approach means that a reference to such state must be created. RTSP uses the concept of a session for this. The session identifier can then be used to access and manipulate the state of a session. Thus, it is possible to utilize several transport connections in a RTSP streaming session, which is suitable in mobile environments where a transport connection may be interrupted in some cases. However, it is recommended to use persistent connections, because else the server may not reach the client. RTSP messages are normally carried over TCP, and the new version of the standard will deprecate UDP as a transport protocol (confer section 6.1.6).

RTSP uses the default port 554. Transport of the actual media data is normally done separate from the signaling. For each available track the server opens a separate data transport stream. Server and client negotiate the ports for these streams during RTSP session setup. The media data can also be interleaved with the signaling protocol if firewalls have to be passed.

RTSP allows intermediate systems to relay signaling messages. Since data transport may be separate from signaling, those *proxies* need not be in the data path. The proxy must then put the destination address of the client into a specific header tag of the session setup message. However, existing implementations may require the proxy to be in the data path for security reasons so that malicious systems cannot direct masses of media data to other systems.

RTSP does not offer requests for handling streaming to a group. Instead, it is supposed that a usual multicast transport address is used for group streaming. The responsibility to handle group streaming is thus placed on lower layers and on applications, where streaming servers may be invited to a conference.

The viewing position is often expressed in Normal Play Time (NPT). NPT has been standardized with MPEG-2 session control [66]. NPT time codes are decimal fractions, which often use seconds as unit. It is however possible to distinguish hours, minutes and seconds in the part left of the decimal point. Thus, `npt=128.3` and `npt=00:02:08.3` are equivalent. Another possibility to express the play-time position is given with SMPTE time codes [117], which have the format

`hours:minutes:seconds:frames.subframes`. Here, the frame rate must be specified, by default a frame rate of 29.97 frames per second is used. In the rest of this work, NPT is used to describe the viewing position.

6.1.2 State

RTSP state is conveyed on a session level. A session can be in one of three states: Init, Ready, and Play as shown in figure 6.1. In each state, only a few methods cause definite state transitions, whereas other methods can be sent without state change. The arrows indicating state changes are marked with RTSP methods, possibly with conditions that have to be fulfilled, and the effects that are caused by a positive answer of the server. Note that redirection is not shown in the state chart for the sake of clarity. The whole set of methods will be described in the next subsection.

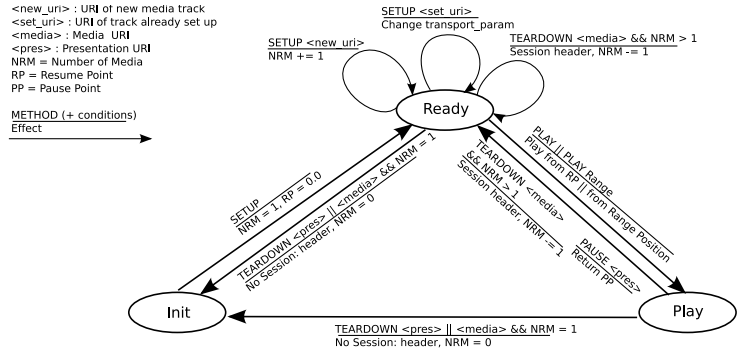


Figure 6.1: Basic RTSP State Machine

In the new version of the standard (see also section 6.1.6), state variables have been introduced to clarify RTSP states processing. These state variables are the number of media in a session (NRM in the figure), and the resume point (RP), which conveys the point in time-line from which a presentation will be played after a pause or jump. These state variables may be set as a side effect of each state transition.

The transitions also depend whether aggregate or non-aggregate control of several media streams is applied so as to handle either the whole presentation or a single media stream of it. With non-aggregate control, a session consists only of a single media stream. If several media should be streamed, several sessions have to be opened. In contrast, aggregate control can include several streams in one session. Though only a subset of the available streams of a presentation may be set up, and streams may be added and removed from a session, control functions always must be applied to the whole requested set of streams. For example, an audio stream alone cannot be muted with a PAUSE, because the video stream would be paused also.

6.1.3 RTSP Messages

RTSP offers a well-defined set of methods. A method is invoked by sending a request message. After processing the request, a response message with an appropriate status code is returned. As

already mentioned, RTSP is a text-based protocol, thus a message consists of several text lines separated by carriage return / line feeds (CRLF) and is delimited by a double CRLF. The method in a request is denoted by a request line, whereas responses contain a response line with the status code and a textual reason. Several header lines provide input for methods or information about specific settings in a server or client. The header lines are distinguished into general, entity and request or response headers. Which header is appropriate for a certain method is subject to method definition.

Though the RTSP standard allows for sending of requests from server to client also, only a relatively small subset of request methods can be issued by a server. This is natural because session-related data is mostly maintained in the server. However, the server may set and query parameters, ask for options, announce a redirection and ping the clients liveness.

Only some methods contribute to state modification, for example DESCRIBE or OPTIONS do not change state. The most important requests for manipulation of session state are as follows: The SETUP request initiates a session and defines transport parameters like destination address and ports. A SETUP request without `Session` header changes session state from Init to Ready. In response to this request, a session identifier is delivered in the `Session` header, which must be used in all further requests concerning this session. The SETUP method is also used to add media tracks to the session and to define and change transport parameters. As such, it can be used in the Ready state, but does not cause a state transition.

PLAY changes the state from Ready to Play and initiates the data transmission by the server. An optional range header can be given to specify the position in the time-line of a presentation where streaming should be started or ended, respectively. For live presentations, the range header value consists of the keyword `now` to tune in the presentation at the current moment. The server response to a PLAY request contains information about the play-time position as well as initial RTP timestamps (see section 6.1.4).

PAUSE stops data transmission but does not delete session state, whereas TEARDOWN removes all state of a session in the server. In order to change the play-time position, a client has to send a PAUSE request followed by a PLAY with the corresponding Range header, because PLAY is not allowed to be sent in play state.

Besides those requests, there are other requests providing for specific functionality. The DESCRIBE requests asks for a description of the presentation which may be sent in the response using a SDP body. REDIRECT allows for changing media data transport parameters, especially for destination address. SET_PARAMETER may be used to set any parameter at the client or server side remotely, and GET_PARAMETER is the method to retrieve parameter values. The standard does not mandate the type and format of the parameters. Mostly, simple text message bodies are used. It is recommended to set only one parameter at a time to allow easier error handling. The OPTIONS request can query information about the server capabilities.

6.1.4 RTP Interaction

Though media data may also be interleaved with RTSP requests, most systems apply RTP as a means of data transport. Thus, the RTSP standard is particularly provisioned to work with RTP. Several parts of the header information may be used to calculate input for the RTP media subsystem and the client player application.

Besides the RTP source identifier, the `RTP-Info` header conveys the *sequence number* and the *timestamp* of the first packet that is sent by the server. The sequence number can be used to detect whether media packets have been sent before or after a control operation. The timestamp forms the so-called wall-clock time and, in conjunction with the `Range` header information, synchronizes the client's play-out time-line with the media frames received. For aggregate control, this information can be used for inter-stream synchronization also. Streams from different sources must be synchronized using RTCP. The RTP marker bit can be used to detect frame boundaries for video streams.

The RTSP standard mandates that RTP timestamps should be independent of the play-time position. Instead, they increase with the wall-clock time by a factor depending on the clock frequency and packetization interval of the media format. For example, during play-out of a 25 fps video stream, the RTP timestamp would increase by 90000 per second, corresponding to 3600 per frame. If a presentation is paused, the wall-clock time still passes. Hence, timestamps are still increased though no packets are transmitted at all. If clients jump in the presentation, no large pause occurs. Thus, only the time required for message transfer and server processing contributes to the increase of the RTP timestamp.

The relation of the Normal Play-Time (NPT) and the RTP timestamp to the wall-clock time during play-out and after control operations is shown in figure 6.2. Here, it is assumed for simplicity that RTP timestamps elapse by a value of 100 per second. A user starts the presentation at time 0 from a viewing position of 0. At 10 seconds, he pauses the presentation up to 20.1 seconds. At this time, the media packet corresponding to NPT 10.1 s is played, which gets the RTP timestamp 2010. The gap in the timestamp corresponds to the pause duration of 10.1 seconds. At 30 s, the user jumps from NPT 20 s to NPT 50.1 s. We assume a small processing time of 0.1 s, which means that the media packet with NPT 50.1 is played out at 30.1 s. Therefore, the corresponding RTP timestamp is 3010. The gap in the NPT reflects the different viewing positions, but RTP timestamps show only a small natural gap.

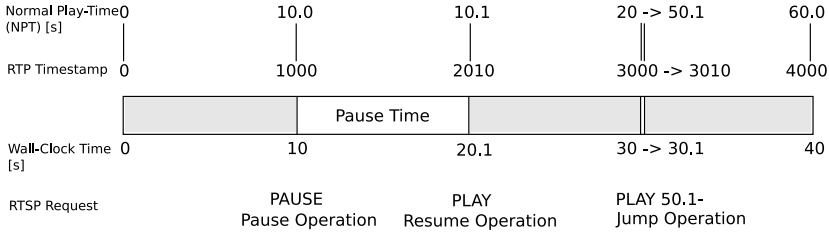


Figure 6.2: Relation of Normal Play-Time and RTP Timestamps

6.1.5 Intermediate Components

If media streaming is intended for commercial success and there are many (in the order of 100,000) parallel requests, intermediate components have to be used, because one server per provider would be overloaded soon. RTSP allows the use of proxies. These proxies may be used for caching so as to serve several subsequent requests for a certain media file.

Proxies according to the RTSP standard may just relay RTSP requests. The standard also defines specific header fields for proxies, which have to be observed by intermediate components. For example, servers can renounce caching.

Another function for intermediate components is sending media data packets to a group of receivers at the same time with several unicast sessions, which is implemented by reflectors. Such reflectors need not necessarily have RTSP functionality. However, clients can signal joining and leaving of the group by RTSP. The Apple Quicktime streaming server may be used as a reflector to serve clients behind network nodes that are not capable of multicast routing [4].

6.1.6 Development and Extensions

Currently, the RTSP standard is undergoing a revision by the IETF MMUSIC working group, with the goal to specify the new version 2.0 of the protocol [146]. This revision is done to clarify, add and remove functionality of version 1.0 according to practical experiences. Unreliable transport of RTSP messages has been removed from the standard but may be defined in an extension. Recording functionality and the announcement of streaming sessions have also been deprecated. Additionally, the state machine has changed with respect to play functionality: In future, a PLAY request is allowed – besides in ready state – in play state also. This means that a PLAY request in play state updates the preceding PLAY request, which is useful to change the viewing position. A foregoing PAUSE request is not necessary any more.

In version 1.0, ports of different media have to be chosen right behind each other, whereas the new standard allows for arbitrarily chosen RTP ports. The IETF working group also made other minor changes to the transport header.

A streaming server as a part of a conference is explicitly mentioned as a use case in the draft standard. Media data are sent to a multicast address or a conferencing focus to be further distributed. However, exact procedures to enable collaborative session control and the distribution of the presentation description are not part of the specification.

RTSP 2.0 explicitly defines a security framework, which consists of authentication mechanisms analogous to HTTP and of transport message protection using Transport-Layer Security (TLS). URIs that apply the `rtsps` scheme identifier indicate that TLS has to be used.

Possible extensions that do not have to be implemented by every server or client are scaling the play-out speed by a certain factor to implement fast forward or rewind operations, and changing the data transmission speed of a server to shorten start-up wait times. In order to group such extended functionality, the feature-tags `play.scale` and `play.speed`, respectively, may be used to ensure interoperability of implementations. Certain features may be required explicitly by a system or queried by OPTIONS requests.

However, functionality useful for collaborative streaming has also been removed from the standard. The possibility to defer media play-out until a certain time by submitting the time parameter in the `Range` header has been deprecated. In our architecture, this parameter could be used to send virtual delays to clients as an optimization to the synchronization modes described in section 4.4.2. Additionally, the `Conference` header which could be useful to establish a logical connection of RTSP sessions to SIP conferences has been removed.

6.1.7 Available Implementations

QuickTime [4] is a settled system, which provides for an end-to-end architecture. It defines an own file format and a rich programming API, which allows for creation, delivery and playback of media. QuickTime uses efficient coding standards like H.264 [72] or AAC [67]. Several application components have been developed as products, whereby the most important for real-time streaming are the Quicktime Streaming Server (called Darwin for the Linux version) [3] and the Quicktime client (for MAC and Windows). Unfortunately, a Linux client is missing. The mentioned interaction with RTP timestamps is not implemented correctly in release 5.5.4. A gap in the play-time position due to a PAUSE/PLAY with range header also causes a gap in the RTP timestamps, which is incorrect according to the standard [146].

A commercially successful streaming system implementation is offered by RealNetworks with the Real/Helix Server [127] and RealPlayer [126]. The company offers several server types dependent on the number of streams it can serve, from a free basic version with 5 streams to unlimited. There is also an open-source version of the client named Helix [125]. The Helix player only plays media coded with open-source formats. Real uses an own media format and adds proprietary headers to RTSP. Additionally, an own data transport protocol (Real Data Transport, RDT) has been developed. Thus, other servers and players are mostly incompatible with Real components.

JMF [154] supports RTSP on client-side. However, the system implements a restricted set of container and streaming formats only. In the FOBS project [170], an object-oriented wrapper for the ffmpeg library, which handles a number of media codecs, has been developed. A plug-in for JMF allows to use those media formats handled by ffmpeg in the JMF application JMStudio. However, JMStudio handles data from file sources differently than data from network sources. Thus, media data play-out only works for data played from a file, whereas data from a RTP source cannot be played, though the JMF RTSP client interacts correctly with the RTSP server. Another Java application is the IBM Toolkit for MPEG-4 [5], which has been originally made for creation and playback of MPEG-4 content. An applet and a stand-alone player which are capable of playing hinted files streamed from an RTSP server are contained also. Developers can use different toolkit API features to adapt these client applications to their needs. However, changing the viewing position is not supported. The license is restricted, and source code is not offered.

Live [93] is an open-source RTSP/RTP streaming library. Some media player clients (VLC [163], MPlayer [123]) use this library for RTSP streaming support. The library can be used for server implementations also. However, the MPlayer client implementation does not use the pausing and seeking functionality offered by the Live library. VLC uses pausing and seeking, but the user interface implementation is not fully functional at that point. The streaming server (former VLS) is now included into the VLC application, but the provision of server-side RTSP requires creation of video-on-demand objects with a telnet interface for each media file, which implies high administrative effort.

MPEG4IP [109] is another open-source project which can be used for RTSP/RTP streaming. Similar to the mentioned IBM toolkit, it emphasizes encoding and production aspects for MPEG-4, by providing a broadcaster, a file recorder, and utilities like an MP4 file creator and hinter. The latter can be used to prepare stored MP4 container files for streaming. There is no RTSP server and proxy available, but the player has an RTSP client integrated, which can be used to play MPEG-4 files from a Darwin server.

The ViTooKi project which examines MPEG-7 meta data descriptions in media production and transfer (see also section 6.4.1.1) includes also an RTSP implementation of server, proxy, and client. However, the system requires to implement MPEG-7 enhancements and is therefore not usable with media presentations that do not include such MPEG-7 meta data. In future, such a system would be very desirable.

In our costream architecture, RTSP client and proxy have to be modified to enable communication with the conferencing and group management subsystem. However, commercially successful systems are hardly able to be modified, whereas available open-source developments often lack some components (proxy, server or client). Streaming libraries are often difficult to enhance with own modifications. Another problem is incompatibility: The interaction with a certain server requires that media frames or samples can be reconstructed from the network packets and that the media formats can be displayed at the client. For proprietary or new formats this may not always be given. It is yet worth to watch the above-mentioned developments, because some of them are improved continually and may offer enhanced features or better media format support in future.

In our costream environment, we use the *beaver* streaming proxy designed and implemented at our institute [8]. Beaver has been developed as a RTSP signaling proxy with reflector functionality, i. e. several client sessions can be coupled to one server session [78]. Transcoding [14] and preferences signaling [58] are other features that have been added. Additionally, it has caching functionality [96] to enhance media data transport for subsequent requests for the same file. An RTSP/RTP client implementation has already been designed in a rudimentary sense.

6.2 Conferencing Systems

In traditional telephone systems, conferencing is an additional feature offered to customers. The H.320 standard offers video-conferencing via ISDN lines [71]. A *Multipoint Control Unit (MCU)* manages all the point-to-point connections of the participants. For telephone systems that run over the IP suite, signaling protocols like H.323 or SIP have been developed for initiation of calls. H.323 refers to a whole family of protocols [159], for example H.225.0 for description of Q.931 call signaling functionality and H.245 as a control protocol for media channels. It uses RTP for media data transport. Like in H.320, conferences are run on MCUs, however these do not have to manage point-to-point connections. The communication is controlled by the Multipoint Controller (MC), whereas media data mixing is done by the Multipoint Processor (MP). The latter is an optional component of an MCU and could also be done decentralized.

SIP [135] has been designed as a more lightweight signaling alternative to H.323. Clients can use basic SIP functionality to take part in conferences, but SIP offers a variety of additional functionality useful for advanced conference management. An overview of the protocol and its functions is given in section 6.2.2.

Since conferencing comprises several other functions besides call setup, for example floor control, membership control or authorization, a general conferencing framework has been defined to enable inter-working of existing protocols and applications. This so-called centralized conferencing framework is described in the following subsection.

6.2.1 Centralized Conferencing Framework

Recently, a working group of the IETF has started to define a framework for centralized conferencing, also referred to as XCON framework [7]. The intention of this framework is to

define a common model for conferencing, which can be accessed by clients using different call signaling protocols. The notion of a conference is tightly coupled with a *conference object*, which represents the conference during its life-time. This conference object holds general descriptions, information about the conference server, the participants, available media, and others. The conferencing system can define several conference *blueprints*, in which default values for the conference object elements are described. A conference instance can then be derived from such a blueprint or from a parent conference instance. *Sidebars* allow to group some participants of the conference to let them exchange alternate or additional media.

Several entities as shown in figure 6.3 may access and manipulate elements of the conference object.

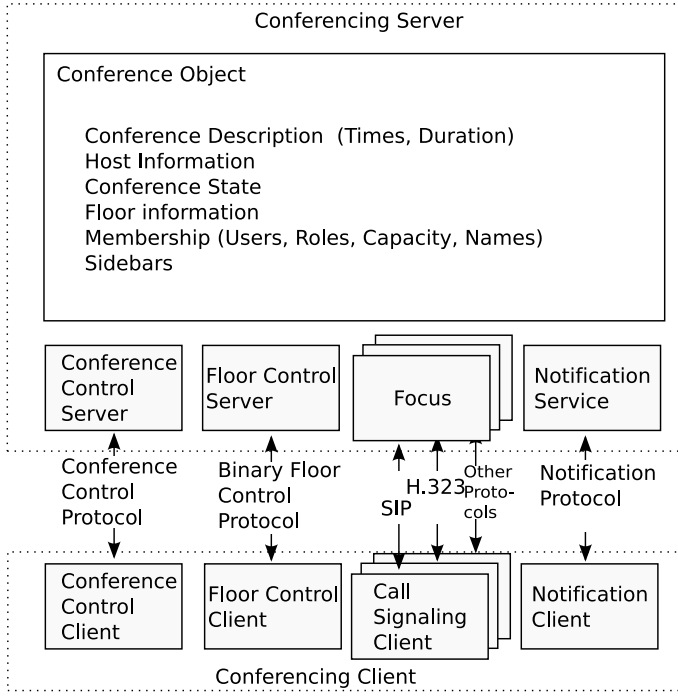


Figure 6.3: XCON Conference Architecture [7]

The *Conference Control Server* allows access to advanced conferencing functionality via a conference control protocol. Some of these features may also be invoked with a call signaling protocol like SIP, which uses a *Focus* to maintain dialogs with all participants. Other call signaling protocols like H.323 may be used within the same conference, but signaling among clients using different protocols may require advanced conference control protocol methods. Changes in the state of the conference object are distributed among users by the *Notification Service*, which can use SIP NOTIFY, for example. Concerning the coordination of participants in a conference to

control a certain type of media, clients communicate with a *Floor Control Server* by using the *Binary Floor Control Protocol (BFCP)* [20].

The data model of the conference object [114] can be seen as an extension to the SIP conference event package (confer section 6.2.2.8). Additional elements are mainly related to floor control. Furthermore, the conference description is enhanced with information about available media. General permissions to handle membership management are another element of the data model.

The XCON framework has been standardized only recently. For this reason, the basic conference control protocol is only defined in an abstract sense, i. e. no specific protocol implementation has been mandated yet. Since the SIP conferencing framework presented in section 6.2.2.7 is compatible with the approach, components of the SIP framework can be implemented and extended later if required.

6.2.2 Session Initiation Protocol (SIP)

The Session Initiation Protocol [135] has been designed since 1996 in order to provide for a lightweight signaling protocol for call or conference (i. e. session) setup, in contrast to the ITU recommendation H.323 [159], which defines a complete system architecture. From the beginning, the intention of SIP has been to provide for personal mobility, i. e. the possibility to manage calls and access services from different terminals. SIP has experienced a great development since then, thus a new version of the Internet standard has been produced together with several enhancements. SIP has found widespread use for IP telephony (Voice over IP), but is a general protocol for session signaling.

With SIP, session state needs not be set up statically but can be initiated when the first participant starts the call or conference. SIP has defined a number of architectural components, which partially can be included into one system. For example, the client, also referred to as User Agent (UA), is composed from User Agent Client (UAC) initiating calls, and User Agent Server (UAS) receiving and processing calls. To handle the inter-working of several domains and provide for intelligent call routing functionality, SIP proxies can be deployed. Mostly, SIP proxies are coupled together with *registrars* that keep address records and location services that resolve addresses of foreign domains. Such systems are often called servers, though the definition of a server is just to receive requests and send responses after processing, which is also the task of a user agent server, for example. In contrast to these *stateful* proxies, *stateless* proxies just relay requests and do not process them.

The functionality of the most important SIP components are illustrated by means of a call initiation example in figure 6.4. User1 residing at Domain A wants to speak to User2, who is located at Domain B. Thus, User1 directs an INVITE request (confer section 6.2.2.3 for details) to the proxy of Domain A (1). This proxy asks its collocated location service for the host that is responsible to process requests to Domain B (2). The location service returns the address of Proxy B; consequently, Proxy A sends the request to Proxy B (3). Proxy B has a registrar which keeps address records of User2 (4). In this case, User2 has registered a device X as the contact for call requests. Thus, Proxy B forwards the request to device X (5). The user agrees to establish a call, thus device X returns a positive response (6), which is forwarded back to the client along the same route of proxies which has been set up with the request. As the client of User1 is informed about the contact device X, further requests as well as the media data transfer can be handled directly without an intermediate proxy (7). These call initiation procedures are described in further detail in the following sections.

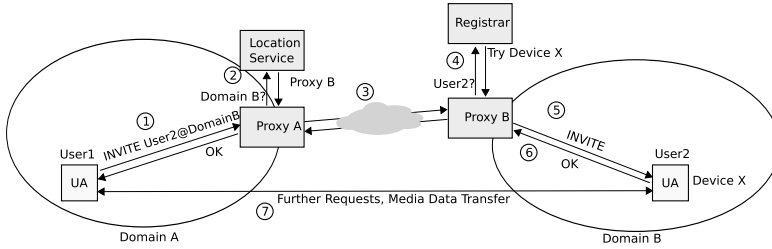


Figure 6.4: SIP Components

SIP has been designed as a layered protocol as shown in figure 6.5. Throughout all elements,

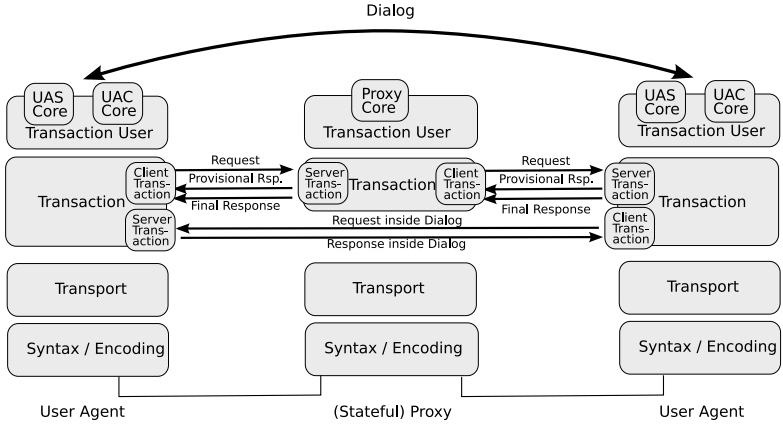


Figure 6.5: SIP Layered Structure

the *syntax* and *encoding* layers are implemented, as well as the *transport* layer. The *transaction* layer is only implemented by user agents and stateful proxies. Above the transaction layer is the *transaction user*, which is formed by the application core of the SIP element. *Transactions* are defined on a per-method base, from the request to a final response. Inside a transaction, *provisional* responses can be sent by a proxy or UAS to indicate that a request has arrived and is processed. Thus, end-systems and proxies take part in transactions if requests have to be routed through a proxy. Transactions are split into server and client transactions, because each of them have own finite state machines to handle request or response processing.

The relationship between end-systems is called a *dialog*. End-systems manage dialog state and keep common dialog identifiers. Normally, proxies do not handle dialog state. However, in certain environments (for example IMS of 3GPP cellular phones [160]) dialog-stateful proxies can be meaningful to enforce policies. This use is controversial among protocol developers, because proxies thus may initiate own requests, which in turn may complicate protocol behavior. In our work, proxies are used for simple call routing purposes, in combination with registrars, and thus do not have any specific functionality.

Each party computes the dialog state at the time of dialog creation, i. e. the server after the generation of the final positive response and the client after the reception of this response. The elements of dialog state and its computation are exemplary shown in figure 6.6. The *dialog ID* is composed of the Call-ID of a message and the local and remote *tags*. Besides, local and remote URIs and sequence numbers are saved. The local sequence number counts the requests the local party has sent, whereas the requests from the remote party are contained in the remote sequence number. The *remote target*, which is generated from the `Contact` header, can be updated with a re-INVITE within a dialog. In contrast to this, the *route set* describing the proxies on the signaling path remains constant, once it has been established by processing the `Record-Route` header. The role of client and server can change inside a dialog. Most methods do not open dialogs, and can be sent within or without an existing dialog.

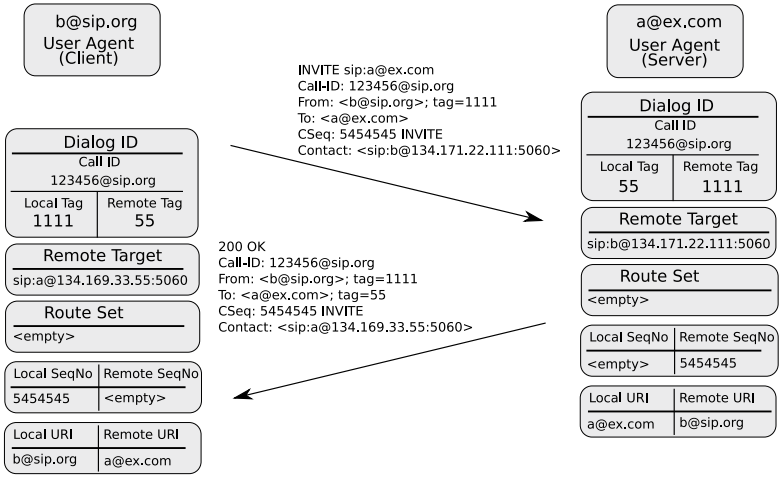


Figure 6.6: Computation of Dialog State

SIP offers a rich set of functions. Several working groups at the IETF standardize these features, not only concerning the basic protocol, but also extensions for specific applications like conferencing or instant messaging.

6.2.2.1 SIP Headers and Response Codes

The SIP standard defines numerous headers and response codes and gives comprehensive information about these. Nonetheless we give an overview of the most important headers, because they are required in any of the methods explained in the forthcoming sections.

From/To denotes the address-of-record of an initiator/recipient of a request. This means that the From and To headers in a response are the same as in the request. From and To can contain display texts.

Call-ID is used as a unique identifier for dialogs and other groups of messages, e. g. registrations. Since they must be unique among all other calls of any other UA, it is proposed that a random local identifier together with the host name is used.

CSeq counts the number of requests within a dialog. Register requests should also be ordered by increasing sequence numbers. Other requests outside dialogs can be numbered freely. Since ACK and CANCEL requests have the sequence numbers of those requests they are related to, the sequence number is accompanied by the method.

Via indicates the list of proxies that a response has to cross. In a request, each proxy adds its own address to this Via header.

Record-Route indicates the list of proxies further requests within a dialog have to travel. Each proxy that requires to stay in the route path adds its own address to this header.

Max-Forwards denotes the number of proxies a request may cross.

Contact is required in INVITE transactions, it indicates a dedicated address a caller or callee should send further requests inside a dialog. An example is the use of a specific host address (instead of the usual domain-specific addresses) and a transport protocol (like `Contact: <alice@host.example.com; transport=udp>`).

Content-related headers describe body data. A `Content-Length` header with a value greater than zero indicates the availability of such a body, which is formatted according to the standard announced by the `Content-Type` header. The `ContentDisposition` header provides for directions how to process the body or parts of the body.

The responses are distinguished by response classes similar to HTTP. 1xx are provisional responses used to stop retransmissions. Normally, they are not sent in a reliable fashion, but a SIP extension (100rel) provides for this. Responses of class 3xx indicate redirections. The client is advised to send another request to the address indicated in the `Contact` header of the response. The classes 4xx – 6xx indicate different types of failure, whereof 4xx are failures which are related to a certain server instance. Often, a modification of the request, like authentication or input error correction, can resolve such errors. The 5xx codes denote server errors, which may be caused by functional problems, for example. In contrast, 6xx codes indicate that the failure concerns the callee user. It is used in case a callee cannot or does not wish to communicate (e. g. using the `603 Decline`).

6.2.2.2 Registration

Users register devices as *contacts* at a registrar. It is possible to register several such contacts, even in parallel, possibly with a preference value. A registrar is mostly collocated to a SIP proxy that is responsible for a certain domain, thus each time a call arrives for a user of this domain, the location service or registration database is queried and the exact contact address is yielded.

Hence, registrations are also referred to as a binding of an *address-of-record* (AOR) to one or several contact addresses. Each REGISTER request must include From, To, Call-ID, and CSeq headers. A Contact header may be given. The following request message shows an example of a REGISTER request. In this case, a user “bob” registers his address sip:bob@example.com using the contact bobhost@example.com with TCP as a transport protocol.

```
REGISTER sip:example.com SIP/2.0
From: "bob" <sip:bob@example.com>;tag=7665
To: "bob" <sip:bob@example.com>
Call-ID: 123456@example.com
CSeq: 4711 REGISTER
Contact: "bob" <sip:bobhost.example.com;transport=tcp>;expires=3600
```

It is important to note that the Request-URI following the REGISTER method (here: `sip:example.com`) specifies the domain of the location service (or registrar) instead of a certain host or user. In most cases, the domain of the location service is equal to the domain of the user. The `To` header is the AOR, in this case `sip:bob@example.com` whose URI has to be resolved to a certain device URI in later requests. Thus, it is only reasonable to register AORs that are routed to the domain of this registrar. Third party registrations are possible and contain the AOR of the initiator of the registration in the `From` header. The Call-ID should be the same for each registration request of this AOR, whereas the CSeq increases by one for each registration request. An expiration time given as parameter of the `Contact` header indicates how long (from the request reception time) this contact registration is valid. In this case, the default value of 3600 s is given.

The REGISTER method is also used to query, refresh or remove bindings. It should be authenticated by proper means [135]. The method itself is not meant to provide for authorization to a SIP server.

6.2.2.3 Initiation of a Session

The initiation of a session is done by a three-way handshake as shown in figure 6.7. The initiator of a session (named the *caller*) sends an INVITE request to the called user agent (shorter: *callee*).

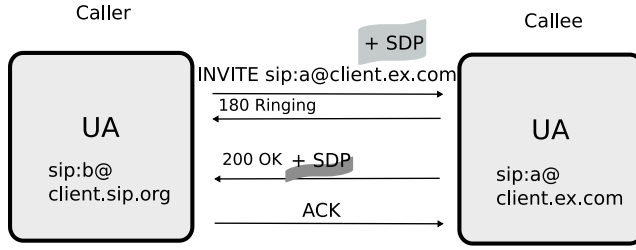


Figure 6.7: Processing of SIP INVITE

Often, there are proxies on the way (1) to route the call to the callee's domain and (2) to resolve the global SIP address into the address of the SIP client at the callee's computer or soft-phone. The proxy may use helper services like DNS, registration data-bases or location services for these tasks. Call routing using a proxy will be described in section 6.2.2.4 in a more detailed fashion. To stop retransmissions of the INVITE, the proxy or the callee can send *provisional* responses to indicate that the request has arrived and is processed, like the `180 Ringing` in the figure. If the call finally is accepted or fails, *final* responses are sent. After a positive final response, indicated by a 2xx status code, the caller sends an ACK in a new transaction to indicate that the call setup is complete. If the final response includes an error code (4xx / 5xx) or a redirection (3xx) code, the ACK is sent in the same transaction, i. e. by the transaction layer. The INVITE method creates a dialog. An INVITE sent within a dialog is used to update information.

This three-way handshake serves as a reliable negotiation of user agent media capabilities, following the so-called offer-answer model [133]. To indicate such capabilities, text bodies compliant to the Session Description Protocol (SDP, confer section 6.4.1.3) have to be added to

the invitation, as shown in the figure: The INVITE and the positive final response carry session descriptions. The first SDP message, intended for session creation, is called *offer*. The recipient, in the figure the callee, may reject certain streams and generate an *answer* as a response to the offer. If an INVITE does not include an offer, the positive final response must include the offer, and the ACK will carry the answer. The offer contains zero or more media streams, with marks concerning the direction (sendonly, recvonly, sendrecv, or inactive), ports intended for reception, and a list of media formats. If several formats are given, a sender can change to a different format in the middle of the session without generating a new offer. The answer must include all information given in the offer to make processing easier. Streams that are rejected are indicated by setting the port number to zero. The direction marks and media formats must match the respective information in the offer [133], e. g. a sendonly in the offer requires a recvonly in the answer.

Since message bodies may be used for other means than offers and answers as well, a `Content-Disposition` header may be used to indicate processing hints for the body. The value "session" is used for session descriptions following the offer/answer model as described above. Other bodies may contain user images that allow to represent the users in a graphical conferencing application. In this case, the type "render" can be used to indicate that the client should process the attached content body. The media type of the content itself is always included in a `Content-Type` header.

In our work, the session description is also used for the transfer of the collaborative group characteristics. If the group communicates via a certain media channel the offer-answer model will be used as described above. This media communication channel may also be set at a later time. In this case, the offer contains zero media streams. To deliver group policies, we expand the offer-answer model of SIP for our needs. In addition to a session description, the description of the group policy is delivered. This is done using a multi-part message body encoded in the MIME format [17]. The format of the policy description will be described in further detail in section 7.4.1.2. Additionally, the presentation description offered by the streaming could be delivered in another part of the message body, because joining clients could test if they support the streaming media formats before giving a response to session transfer. However, this would lead to considerable overhead, because most clients access presentation descriptions during the RTSP signaling procedure. We therefore assume that each client supports at least some of the available media formats of a streaming presentation and do not include the presentation description.

6.2.2.4 Call Routing

As already mentioned, the caller may route all requests through a proxy which discovers the appropriate route to the callee's domain (by using the Domain Name Service (DNS) [134]). Users can be called on different client devices, each with its own address. A caller does not have to know the specific device address, but can use the registered SIP URI of a callee at the registrar responsible for the callee's domain.

After dialog initiation, further requests may be sent directly between user agents, because user agents include their device addresses in the `Contact` headers of dialog-initiating requests and responses. Proxies may, however, require to stay in the routing path of subsequent requests by including `Record-Route` headers. This functionality may be useful for mid-call features like call transfer or billing, however it is critical regarding performance and scalability.

A simple example of call routing is shown in figure 6.8. In this example, the UAC sends a dialog-creating request to the proxy of its domain (which has been pre-configured through means

outside SIP). It uses a `Route` header with the *loose routing* (`lr`) parameter coupled to the URI. Loose routing means that the proxy conforms to SIP Version 2.0 and handles the destination separate from the list of proxies that have to be visited. The proxy of the sip.org domain adds a `Record-Route` header containing its own address, thus it will remain in the routing path for all subsequent requests. It sends the request to the proxy of ex.com, which in turn is responsible for routing to the right device of the callee. Hence, the registration of a@ex.com, which resolves to a@client.ex.com, is used as the new Request-URI. The client finally answers the call by sending the response back to the first host included in the `Via` header list. Thus, the response traverses all the proxies which have seen the request. Each user agent updates its route set from the `Record-Route` header list at the time of dialog creation. Subsequent requests within a dialog thus travel to exactly those proxies included within this route set, in this example only proxy.sip.org, since proxy.ex.com did not require to be traversed.

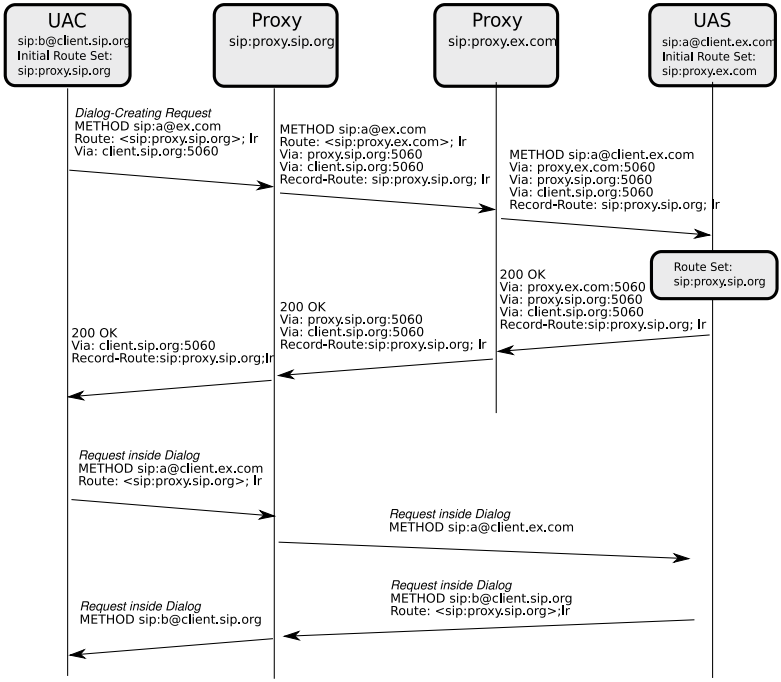


Figure 6.8: Routing of Requests in SIP

6.2.2.5 Event Notification Framework

The event notification framework of SIP has been described in an additional Internet standard [130]. It should not be considered as a general publish/subscribe framework (see also section 4.2.5.2), however in an environment where SIP is already used it may be valuable to deliver information about distributed events. User agents interested in receiving information send SUBSCRIBE requests with the URI of a certain resource to a server that is capable of generating

NOTIFY requests. In the `Expires` header, the duration how long notifications should be sent is given. The notification server may terminate a subscription before this expiration time. If an expiration value of zero is requested, only one notification about the current state is sent and the subscription is terminated at once. The subscriber additionally specifies the format which the body should be compliant to in the `Accept` header. Below, an exemplary subscription is shown: Alice subscribes to the conference state at group0. She wants to be informed with a document conforming to the *conference-info* XML format and likes to keep up her subscription for 3600 seconds relative from the time the subscription has been added in the notification server.

```
SUBSCRIBE sip:group0@example.com SIP/2.0
From: "Alice" <sip:alice@example.com>;tag=8037
To: "group0" <sip:group0@example.com>
Contact: "Alice" <sip:alice@siphost.example.com;transport=udp>
Expires: 3600
Accept: application/conference-info+xml
Event:  conference
```

If such a notification server accepts a subscription with a 2xx response it also generates a NOTIFY request containing the current state of the resource in the body of the NOTIFY. The event type as well as the content type are included as request headers. Besides, the state of the subscription is given: With an active subscription, the user will be notified of further state changes. Additionally, the remaining time of the subscription is included in the `expires` parameter. In contrast to this, a terminated subscription state means that no further notification will be sent.

```
NOTIFY sip:alice@example.com SIP/2.0
From: <sip:1138721171701@example.com>;tag=6284
To: "alice" <sip:alice@example.com>;tag=2519
Expires: 3600
Event:  conference
Content-Type: application/conference-info+xml
Subscription-State: active;expires=3600
Content-Length: 898

(body not shown here)
```

If an event generator is logically separate from a notification service, event state may be sent to the notification server by using the PUBLISH method [113].

The format of this body and the underlying data model are specified in a so-called *Event Package*. This event package also specifies the name of the event, the possibilities to use partial (i. e. incremental) notifications and the frequency of notification delivery. Event packages can be defined for arbitrary state information.

Among the available event packages is the *presence* event package [131], which allows to subscribe to the presence state of a certain user. Especially for instant messaging applications, this allows users to keep informed about the state of their contacts. Another important package is the *conference state* event package [137], which will be described in further detail in section 6.2.2.8. The notifications of this package distribute conference information and state changes with respect to member set, services or available media types.

In our work, membership state corresponds to the conference state and is delivered with the conference state event package. Additionally, the *dialog* event package [138] is used to find out

the identification of a collaborative group from the dialogs of a conference server or a certain user, in case a pull initiation transaction should be used. The presence event package can be used to distribute information about whether a user is online. This package can be seen as a convenience package and less existential for the functionality of our architecture.

6.2.2.6 Call Transfer

A different method for call control is call transfer. Calls can be transfered to users as well as to user devices by using the REFER method [151]. The recipient of this method should try to access the resource given in the Refer-To: header. This may be an arbitrary URI. If the resource is a SIP URI, a SIP request to this URI according to the given method is sent. In the example shown below, user Alice sends such a REFER to *group0*, which should invite Bob to the session.

```
REFER sip:group0@example.com SIP/2.0
From: "alice" <sip:alice@example.com>;tag=8331
To: "group0" <sip:group0@example.com>
Contact: "alice" <sip:127.0.0.1:5072;transport=udp>
Refer-To: "Bob" <sip:bob@example.com; method=INVITE>
```

The recipient of the REFER should check its own policy if the REFER may be accepted. If this is not the case, a decline response is sent back. Acceptance of the REFER does not necessary imply that the referred method succeeds. Therefore, feedback of the success of the referred operation should be given to the sender of the REFER. The use of the REFER method also implies a subscription to the *refer* event using the above-mentioned event notification framework. The recipient of the REFER request creates a subscription and distributes notifications of the status of the reference. Here, the REFER method is used like a SUBSCRIBE method. The agent processing the subscription may decide how long the subscription is valid. The SUBSCRIBE method itself cannot create a subscription to the refer event, but it can be used to terminate the existing subscription. The agent sending the REFER may also terminate the subscription immediately without influence on the call to the third party itself. Subscriptions may also be terminated by the notifier, i. e. the client the REFER has been sent to.

An additional feature that is currently under consideration is the use of the *multiple* REFER option [19]. This is especially useful in conferencing scenarios, for example if a conferencing server should send a certain request to a number of participants. Following this option, a REFER request can contain a pointer to a list of multiple REFER targets. The recipient of a REFER should then execute the specified method for each of the list entries.

6.2.2.7 Conferencing Models with SIP

Since SIP aims to offer similar call control functions like in usual telephone systems, conferencing with three or more parties as an extension to two-party calls is another interesting application for SIP. The basic protocol provides for SIP dialogs, which can obviously manage two-party calls. However, in the multi-party case the communication between all participants must be set up and managed.

One possibility, the so-called “loosely coupled” model, would be to use a multicast group for communication. Though this can provide for a very efficient data communication, dependent on the routing protocol, no specific signaling relationships exist in this case. Users can join or leave the communication by joining or leaving the multicast group. The only purpose of signaling

would be to announce the multicast group addresses, but it is not possible to gain knowledge about the state of participants in this case without additional mechanisms, e. g. to retrieve the state from feedback on media data transport using RTCP (confer section 6.3.3).

Another obvious method would be to set up dialogs between all participant pairs. This leads to a fully distributed communication, which would require $n(n-1)/2$ dialogs for n clients and thus would not be very efficient for larger groups. Furthermore, it is more difficult to manage a consistent state between all participants.

To overcome the difficulties of existing models, the SIPPING working group has developed a framework for so-called tightly coupled conferences [132]. Like in the compatible XCON framework, these are characterized by the use of a central point of control that every participant can connect to. The advantage of this *focus* is the easy maintenance of a consistent conference state and the deployment of media mixing functions, which enable a more efficient data transport. The focus is a logical entity which can be physically implemented in several fashions, either on a central server or collocated with a user agent. A media mixer can reside on a centralized focus for smaller conferences or can be implemented in a distributed fashion for larger groups.

The SIPPING working group suggests a number of different call flows for conference control [77] as best current practice. The initiation and termination of a conference as well as the addition and removal of members can be handled with usual SIP messages like INVITE or REFER. A possibility to create ad-hoc conferences is the use of a *Conference Factory*, which returns the created URI in a redirection response to an INVITE.

The SIP conferencing framework also allows sub-conferences within a conference. Within such a *sidebar*, members can open a separate audio channel, for example, to have a discussion that cannot be heard by other members. Sidebars can be treated like parent conferences [37], but cannot exist without parent conference. Until the time of writing, no SIP-specific call flow to establish a sidebar to a conference has been defined. Instead, it is assumed that non-SIP means are used to manage sidebars. However, since a sidebar has an own URI that is mapped to the focus, it is possible to establish a parallel signaling dialog to the sidebar and use the same call flows for sidebar conferences as for parent conferences.

The central control point model is a feasible way to establish conferences in a tightly coupled sense as it is also required in collaborative streaming scenarios. Even user agents that are capable of basic SIP only can participate in these conferences. However, for collaborative streaming it is reasonable to have conferencing-aware clients which can be equipped with SIP extensions like the REFER method.

6.2.2.8 Conference Event Package

To distribute events concerning state of a conference, for example joining and leaving of participants, the *conference* event package has been designed [137]. It can be used for tightly coupled conferences as described before. Users subscribe to a conference URI, and are informed about state changes by the focus that represents the conference (or its associated notification service).

The conference state that is reported in NOTIFY requests is defined as an XML document. It comprises changes in membership in the `<users>` element, in the status of participants by the `<user>` element, in conference and service URIs (by the `<conference-info>` and `<conference-description>` elements), and possibly other state information (like sub-conferences). Each of the mentioned elements contains sub-elements to provide for more detailed information.

The conference event package also enables to generate different views dependent on subscriber roles and to protect privacy of users by hiding their identity from other subscribers.

The conference event package can be used by collaborative streaming systems to convey group membership state. Associations can be reported as sidebars to the conference. For the association state, additional elements have to be defined, or a specific package has to be developed.

6.2.2.9 Available Implementations

With the upcoming success of IP telephony, many commercial SIP implementations are available, for example SIP phones, user agent applications, application servers, or development toolkits. A number of service providers offer SIP services like Internet telephony, sometimes in combination with geographic numbers or connections to the PSTN.

However, researchers also implemented open-source libraries, sometimes as spin-offs of commercial products. Most of these libraries offer basic SIP functionality and are compliant with the new standard RFC 3261. OPAL [150] is the successor of the OpenH323 implementation and supports both H.323 and SIP. The developers claim that other protocols may be added easily. Programmers can use this library for pure SIP applications also, but additional class libraries are required.

The VOCAL SIP stack [165] provided by Vovida includes a SIP stack as well as application service building blocks. It has not experienced any development since release 1.5.0 in 2003 and does not compile on newer Debian Linux machines, thus it is not considered any further. ReSIProcate [129] also offers a stack and reference applications, for example a SIP proxy and registrar named *repro*. It is an open-source library which is intended to be used in private or commercial applications. The object-oriented design in C++ allows to derive applications adapted to own requirements. As an interesting feature, a Dialog Usage Manager provides a higher-level programming interface for user agents.

The eXosip library [142] written in C provides a higher-layer API for the osip library [105], which provides for low-level parsing and message handling functionality. In turn, eXosip allows to use convenience functionality to initialize and maintain calls. However, this API does not offer dialog management, thus the programmer must write own data structures for saving information connected with the call.

The nist-sip project [118] offers a richer set of functionality, and keeps to the layered protocol architecture chosen by the standard. Programmers normally only need to build their application core on top of the library and fill messages with appropriate values. A drawback can be seen in the fact that the implementation is done in Java, which lacks speed compared to a C implementation (see also section 9.2, where the latency of the SIP message flows of our architecture is evaluated).

6.2.2.10 SIP Functionality in this Thesis

Among other functions, the basic call signaling of SIP is applied in this work. Proxies may be used to handle call routing and registration, which is convenient for those clients which do not resolve names with DNS on their own. However, we do not examine proxy routing issues like forking in our architecture.

The most important extensions that are used in our architecture are listed in the following summary:

- Parts of our architecture are based on the conference framework, because it provides a model for tightly coupled conferences. We also conform to related specifications like the standard on call control message flows [77] whenever possible.
- The REFER extension, originally intended for call transfer, is used to push sessions and, in turn, add group participants. Aside from the standard, we use REFER also to register streaming presentations to the collaborative group.
- The event notification framework is used to collect information about groups, streaming sessions, and participants. Hence, we also make use of the conference and dialog state event packages. To deliver streaming session state, we must design an own package so as not to adversely affect existing parsers.
- The multiple-refer option is used to invite a group of participants to a conference. Although such a list could also be given inside an INVITE request at the start of the conference, we have chosen to use REFER, because this request can be used at any time during group lifetime.

Some other mechanisms have been enhanced a little to fit into the context of collaborative streaming: Session descriptions compliant to the offer/answer model are packed with a description of the chosen conference policy into a multipart body.

There are numerous other features that will be used in real-life conferencing applications, but are secondary to collaborative streaming: For example, we do not emphasize media negotiation and use zero media offers and answers only. We also did not implement preferences, capability negotiation, third-party call control and transferring two-party calls to conferences by using the `Replaces` header. Future work can enhance our conferencing application with these features and thus serve as a comfortable collaborative system.

6.3 Media Data Transport

Since streaming media is no completely new concept, much effort has been spent for research about efficient transport of media data (confer to section 3.1.2 also). Media data are continuous data, thus there is a relation in time between the frames or samples of a media stream. In many media streaming scenarios where interactivity is only a secondary concern, it is less important that a specific end-to-end delay from sender to receiver is met, but possible delay variations (jitter) should be kept small and hidden from the application. To enable this from a networking perspective, QoS mechanisms must be applied to provide for a certain delivery guarantee. A lot of proposals exist in this area (like the Integrated Services [12] or the Differentiated Services [11] architectures), but are not widely implemented. For the work proposed in this thesis, QoS mechanisms are not further examined, because they do not influence the design of our architecture.

Requirements for media data transport protocols include:

- Frames must appear in the right order, or the application must get a means to restore the correct order.
- A certain fraction of lost packets is tolerable, depending on the media format.
- Frame boundaries should be consistent to packet boundaries to minimize the perceived loss.

In principle, TCP can be used for transport of media data. The data stream is handed to the application in the correct order, without loss, thus media frames can be regenerated completely. Exact synchronization of different tracks is not possible, thus the transport stream contains all presentation tracks. Congestion control is implemented in TCP, which is useful for wide-area networks. However, it is difficult to use a time relation for transport, which is a reason that TCP is often considered as unacceptable for streaming media. Reliable transport can, on the other hand, be useful for connections from servers to caches.

UDP alone is not sufficient for media data transport, because some features like sequence numbers, timestamps or frame border markers are missing. Hence, a new protocol which provides for these features has been developed and published first in 1996.

6.3.1 The Real-Time Transport Protocol (RTP)

The Real-Time Transport Protocol [144] is available in version 2 since the year 2003. Despite its name it is an application-layer protocol which uses UDP as a transport protocol. It is designed as a protocol which gives a framework for transmission of real-time data, which need not be audiovisual data. For a classification into types, *profile* definitions are used, e. g. the AV profile handles transport of audiovisual data.

RTP uses the concept of application-layer framing to keep packet boundaries consistent to coded media data portions. This is advantageous in case of loss, because the number of affected frames is reduced. If several packets need to be send to cover one frame, a marker bit is set to indicate the frame boundary. RTP uses sequence numbers to allow for detection of lost or re-ordered packets. Timestamps in the packets give the timing order of media frames or samples. The actual handling of these protocol features (e. g. how packetization must be done) for specific media formats is defined in the *payload-type* definition standards.

Quality of service issues are not supported directly in RTP, but rather in the companion control protocol *RTP Control Protocol (RTCP)* where sender and receiver reports are exchanged which allow the collection of statistics. However, reactions on any reporting events have to be done by the application itself. Additionally, there have been many attempts to support reliability in RTP and RTCP, like sending a list of lost packets challenging a retransmission of these.

For controlling the rate of applications, approaches that are *TCP-friendly* have been considered important. TCP-friendliness means that the applications should react to congestion situations with a rate adaptation, similar to what TCP does. Thus, the *TCP-friendly Rate Control (TFRC)* [51] can be used with RTP, possibly using extension headers [46].

In our further implementation we use standard RTP and do not rely on any additional support. However, the choice of a specific RTP flavor does not have an impact on our architecture.

6.3.2 RTP Timing

RTP timestamps are basically computed out of the frame rate or sample rate of the application. The timestamp must be initiated by a random positive integer number, chosen independently for each RTP stream. The increment rate is then for audio the packetization interval (how much of the audio is contained) times the sampling rate (how often are the values sampled). For example, the timestamp for each audio packet increases by 320 if these packets contain 20 ms of audio sampled at 16,000 Hz. If the frame rate of a video stream is fixed, the video timestamp increases

by 90000 divided by this frame rate, otherwise the timestamp must be calculated from the system clock.

The role of the RTP timestamps is to place frames in the correct timing order. The application can then perform intra-stream synchronization. Since video frames may be sent in several RTP packets, all with the same RTP timestamp, sequence numbers have to be in place to detect losses.

The synchronization between media (inter-stream synchronization) is done using the so-called NTP timestamps in the RTCP Sender Reports. This timestamp explicitly relates the RTP timestamps to a common wall-clock time. If streams from several sources are to be sent, these sources have to be wall-clock-synchronized, otherwise receivers just have to synchronize to the sender. In RTSP, the relation of the RTP timestamps to the wall-clock time is provided by specific header information, thus streams from a common source do not have to be synchronized using RTCP.

For media streaming, it is especially interesting how RTP timestamps are dealt with in case of gaps in play-out, i. e. with pauses or jumps in the presentation. RTP timestamps must increase in a continuous and monotonic fashion and can be seen as a wall-clock time that is passing. Thus, during the pause time the timestamp will also pass, i. e. after a pause there will be a certain gap in the timestamp, and after a jump the timestamp will just continue, i. e. there may be no gap in the RTP timestamp. This is done to ensure that the RTP media renderer is not affected by such jumps, otherwise the process could spuriously handle packets after a jump as duplicates.

6.3.3 RTP Control Protocol (RTCP)

RTCP is used to exchange information about media data transmission. It is a means to get information about data transmission on an unreliable transport channel. Another purpose is synchronization of independent RTP media streams as mentioned above. It conveys a number of message types to be sent in sender and receiver reports. These reports are sent with a certain controlled rate that derives from the number of participants within a session. The total amount of bandwidth used by RTCP messages is 5 % of the session bandwidth by default, however it is possible to specify different values independent of session bandwidth [21].

The sender and receiver reports are designed to give feedback on the quality of data transmission. This supports the tasks of flow and congestion control at the sender as well as third-party monitoring of network providers. By transmitting the canonical name (CNAME) of participants, these can be tracked. As already mentioned, RTP timestamp and a corresponding NTP timestamp have to be given. The RTP timestamp need not be the timestamp of a packet actually sent, but should be calculated as if a packet had been sampled at the instantaneous time, derived from NTP-synchronized wall-clock time or system clock time.

The NTP timestamp expresses seconds relative to 0 h UTC on January 1st, 1900. It is a 64 bit fixed point number value, the first 32 bits provide the integer value and the second 32 bits the fractional value.

The sender report (SR) contains sender packet counts and octet counts. Receivers can then determine lost packets and give their fraction of lost packets back in a receiver report (RR). The BYE packet serves as an indication that a participant has quit the media channel. However, due to the unreliable nature of RTCP, applications should implement fall-back mechanisms to safely end a transmission or reception. The last packet type, the so-called APP packets, can carry arbitrary application data.

6.3.4 RTP Intermediate Components

RTP has mainly been designed to serve for audiovisual conferences. In order to efficiently use network resources in larger conferences, intermediate components have been designed to manipulate the data streams. The *Mixer* is used to merge streams from several sources into one stream. In our architecture, this is useful for an optional discussion channel.

A *Translator*, which is a RTP relay that transforms one RTP connection into another, can serve different purposes. First it can be used to establish a tunnel through a firewall, with translators relaying to and from a secure point-to-point connection through the firewall. It also can serve as transcoding unit for single streams without mixing them, which is useful for media delivery to heterogeneous devices in collaborative streaming.

6.4 Configuration of Multimedia Services

Multimedia services are often used for entertainment purposes. This means locating service components and setting of service parameters to personalize the service should be done without much user intervention in order to provide for a convenient and simple user interface. For personalization, description of meta-data is required: First, the media presentation or session must be described, second, the user preferences must be described.

In this section, related protocols and definitions to describe presentations and sessions and user preferences are presented. Besides, we give an overview of service location protocols.

6.4.1 Description of Session Meta-data

Meta-data are defined as “data about data”, they deliver technical or content-related descriptions. Such meta-data are helpful regarding several aspects: First, to search content after given keywords, second, to distinguish different occurrences of content after specific properties, like resolution or language.

Such meta-data descriptions must be delivered by content providers as well as users: content providers must specify the properties of presentations, whereas users must specify their preferences. It would be reasonable if a common language would be used for both.

6.4.1.1 MPEG-7 as a Generic Approach

The Multimedia Content Description Interface, better known as MPEG-7 [108] serves as an interface for semantic and structural description of audiovisual data. This provides for a universal indexing during the whole lifetime of the content, from creation to content search and transport of descriptions, in contrast to foregoing standards.

The standard consists of eight parts, which all in all can describe the whole process of generation, transport and archiving of meta data. There are so-called *Description Schemes* to explain the relation between *Descriptor* components. Those Descriptors describe characteristics of audio or video data. By help of a structural language, the so-called Description Definition Language (DDL), which is based on XML schema, new description schemes can be created. Generic tools for description are defined in another part of the standard.

A number of implementations are available, e. g. [116], some of which even automate and try to unify the process of meta data creation from the audiovisual data themselves.

MPEG-7 deployment in currently available media applications would be useful to help search engines to find appropriate content. Today, service providers either implement proprietary tools or rely on multimedia hardware vendor standards. For example, the audiovisual part of UPnP (UPnP AV) [42] offers a Content Directory for retrieval of audiovisual content according to search requests.

For collaborative streaming, generic descriptions like MPEG-7 can be useful in the sense that each user can retrieve presentations from his/her own media service provider as shown in figure 6.9. It is sufficient that group members like the clients A and B in the figure use a common collaborative service, whereas media streaming can be offered by different services. Clients A and B direct their streaming request with a meta-data description of own requirements to the streaming services offered by the respective service providers, possibly with synchronization information from the session sharing service.

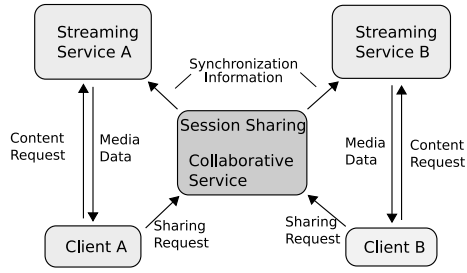


Figure 6.9: Common Collaborative Streaming with Different Media Providers

6.4.1.2 SMIL

The Synchronized Multimedia Integration Language (SMIL) [167] has been developed to integrate different media objects (text, graphics, audio, video) into one continuous presentation. The presentation and the objects that contribute to it can reside on the local system or in the global Internet. Similar to HTML, links can be given in the presentation to enable a kind of interactivity for the user.

In the context of collaborative streaming, SMIL can contribute to individual flavors of the same content: Different users may choose different languages, objects of different bandwidths can be chosen to aid mobile systems, or the layout can be adapted to player capabilities. Since RTSP URIs can be given in SMIL expressions to retrieve streaming presentations, SMIL can just be used as a front-end description language in collaborative streaming systems. The collaborative streaming architecture is not influenced by the decision if a presentation is retrieved directly using RTSP or via a SMIL player where the RTSP presentation is embedded.

6.4.1.3 Session Description Protocol in Streaming and Conferencing

The Session Description Protocol (SDP) [52] has been developed for announcement of sessions originally. SDP is no usual request-response protocol, but a structured textual description following a rather simple type-value principle. The description consists of lines, which are type fields followed by a = and a value. The lines again are separated by end-of-line characters.

Common SDP type fields are the *owner* or creator of a session together with a session identification (`o=` field), the *session name* (`s=` field) and different informative fields like e-mail, description URI, or phone number, which can be used to contact a person responsible for a session. Textual *information* can be conveyed using the `i=` field. The type fields should not be extended, instead, the *attribute* field `a=` can be used for application-specific means like shown below.

Since descriptions have found to be useful in a number of other applications, like interactive session setup or exchange capabilities, the application range of SDP has broadened. Thus, on the one hand, several extensions of SDP have been defined, on the other hand, IETF multimedia protocol standards using SDP define attributes that make specific use of descriptions, for example the `a=range` attribute. An extension to SDP itself is e. g. the grouping of media lines according to specific characteristics [18].

The use of SDP in streaming and conferencing, as well as in media data transport, shall be described here, since it is important for our example implementation.

SDP descriptions are conveyed by streaming servers in response to a client request, either by a RTSP DESCRIBE or by HTTP. Descriptions of streaming presentations include the following items:

Media streams enumeration with a media line

```
m=<media-type> <port> <transport protocol> <format>
```

and several media specific attributes (using `a=<value>` lines). All attributes following a media line are so-called media-level attributes and are assigned to this media line. Session-level attributes, in contrast, are those attributes preceding all media lines. The following media information have to be used particularly for media streaming:

Media type: Video or audio, or other type of media.

Transport parameters: Transport information like protocol and port is conveyed here, though this is mostly conveyed in the transport header line of a RTSP SETUP request or response. However, a client is recommended to use the port that is indicated herein.

Format-specific parameters: Format parameters contain a static or dynamic payload type. If a dynamic payload type is used, a mapping attribute (`a=rtpmap:`) must assign a well-known type name to this number. If client applications need more information to decode and display this kind of payload data, they have to be conveyed in this attribute (`a=fmtp:`). The actual contents of this line are defined in standard documents concerning the transport of this specific payload type.

Control URI(s) conveyed by the `a=control:` attribute. If such a control attribute is present on session level, aggregate control can be used. Still, control URIs for single media tracks can be available. For non-aggregate control, control URIs must be available on media level only.

Presentation range concluding the period between start and end of the presentation. The `a=range:` attribute together with different time formats like normal play-time (NPT) or clock is used therefore. The presence of a start time of zero and no end time indicates a presentation that is not searchable. Thus, only pause control operations can be used.

Time of availability which is mostly relevant for live presentations.

Connection information indicating the destination address to which the presentation should be sent with a `c=` line. In unicast sessions, where a client specifies its address within RTSP message exchange, this address should be set to a zero address (0.0.0.0 or 0:0:0:0:0:0 for IPv4 or IPv6, respectively).

In conferencing with SIP, SDP has similar functions. Participants of a conference specify their capabilities for media sending and reception in the body of a SIP INVITE or ACK, following the offer-answer model for capability negotiation [133]. The media type fields are used similarly as in RTSP descriptions.

SDP is not a complete meta data description. In the previous version, the language of audio tracks could not be described in a standardized fashion, for example. Thus, applications like streaming servers did not include such meta data in their media file descriptions. However, the IETF has specified a new version of the standard [52], where several new attributes are standardized, for example keywords or media languages. Additionally, new attributes are registered in their own standards documents, for example the use of label or content attributes. Still, many presentations are described according to the old version only.

In this work, SDP is used for description of the session because of its widespread use for streaming and conferencing in the Internet. It is expected that media servers either enhance their meta data according to the new SDP standard or follow other standards like SMIL or MPEG-7. For collaborative streaming, attributes like keywords and languages are sufficient, thus we do not contribute further to meta data descriptions.

6.4.2 Preferences Description

Preference description has been an issue in multimedia signaling for telephone calls and conferences based on SIP. In an extension to the SIP standard, the management of caller preferences is described [136]. Such preferences are considered essential to establish a reasonable conversation between caller and callee. Instead of only specifying own capabilities in SDP descriptions, definite requirements for calls can be expressed. The `Accept-Contact` and `Contact` headers of SIP requests are used to express such preferences by tag/value pairs. A `Reject-Contact` header is introduced to describe negative preferences. Besides that, it is possible to express hard constraints by adding further feature tags like `require` and `explicit` to these headers. To order the preference sets, numerical q-values are used.

In [80], a framework for signaling preferences with an advanced management is presented. The authors propose not to relate whole feature sets, which have to be matched with a complicated algorithm, but to express a base preference with a partial order on single features. The complex user preference can then be constructed by combination of these features. The authors propose to use Pareto accumulation and preference prioritization operators to enable this. Since considerable resource effort may be spent to signal such user preferences, preferences should be reduced by proxy components as far as possible.

User preferences with streaming servers have been considered as less relevant, since it is considered the problem of the user or client how to deal with what the server offers. Little research work has been spent on choosing an optimal service out of preferences the user provides and capabilities the server offers. The capability and preference description language CC/PP has been used for a signaling system of user preferences in the transcoding context, which can be used with

a RTSP SET_PARAMETER message at a RTSP proxy with certain (transcoding) capabilities [58].

In our work, preference signaling is partly relevant. It may be useful in discovery of content, users and services. If we state that the number of collaborative streaming services, or components that serve for parts of a collaborative streaming services, may grow with time, different implementors may provide for different services. Thus, it is reasonable to support the user's choice for an appropriate service. However, we concentrate on signaling preferences for group policies rather than for usual multimedia features.

6.4.3 Service Location

In our collaborative streaming system, service location includes three aspects: First, an Internet streaming service that offers a certain content has to be located. The service location mechanism thus should offer searching by titles or keywords. Second, devices in home or mobile environments that run collaborative clients must be found. This also means that service location must adapt to possibly changing network environments. The third issue is to locate the collaborative streaming service as an intermediate system. This intermediate system may also be pre-configured in some scenarios. Additionally, if the collaborative streaming service is implemented in a distributed fashion, the service components must be configured with the addresses of each other.

Several service location protocols have been developed by different consortia. The goal of these protocols is mainly to help non-expert users with the configuration of the application. The protocols presented below mostly emphasize their own aspects like device configuration, scalability or dynamic registration. In the following, SLP, Jini, and UPnP are introduced as representatives of service location protocols. Many other protocols like Bluetooth SDP [148] also provide for service location features, but are partly restricted to closed networking environments and thus not considered here any further.

6.4.3.1 SLP

The Service Location Protocol [50] is an approach targeted to enterprise-level discovery of services like printers, web services etc. Service Agents (SA) can provide information about a specific service. User Agents (UA) multicast requests for services in a certain network. The SA responsible for the requested service will then give a Service Reply providing a link on further information. Using optional requests, it is also possible to query all services of an agent or certain attributes. Besides, UA can apply search filters conforming to regular expressions like LDAP [169] filters.

Each service must have an URL to distinguish it from other services. This URL specifies the service type, which is either an abstract type like `service:printer:` or uses a schema name from a well-known protocol like `http:`. Additionally, the service can have a number of attributes, which are key-value pairs. These attributes describe service characteristics and configuration information for the clients. Clients can submit requests to find services which have certain attributes. Service templates [49] are used to specify the attributes used by a certain service to enable interoperability. Such a service template exists for SIP to find registrars or outbound proxy services, which use certain transport protocols, for example.

The service must reside in a certain *scope*. Scopes are simple strings and are used to group services. A request submitted within a certain scope will not find services which are grouped

to different scopes. Hence, administrators may restrict users and services to certain scopes to allow more precise lookup. To optimize service discovery, Directory Agents (DA) may be implemented which handle the information for several SA. Experimental extensions to the SLP, the so-called *mesh enhancements* enable the inter-working of several such DA to support automated registrations in larger networks [174]. Additionally, *preference filters* can be used to find the best match of several services and thus save bandwidth for search results. *Global attributes* enable to find services of several types with one search query [173].

6.4.3.2 Jini

Jini [156] has been invented to provide for location and access of distributed resources, thereby it is suitable for distributed, ubiquitous computing systems. The central component is a so-called *Lookup Service* where services are registered. Clients discover this Lookup Service and query it for the service they want to use, possibly limited by some attributes.

Jini provides a specific mechanism to access discovered services derived from the Java *Remote Method Invocation* (RMI) mechanism. A (bytecode) service object is sent to the client to serve as a proxy. Services can then be accessed by executing this bytecode. This has the advantage that programmer interaction is not needed for each service configuration, but instead a defined RMI interface is used. The disadvantage can be seen in security problems, because access controls have to be defined for both client and server side. Jini relies on the Java 2 security infrastructure. Without additional mechanisms like authentication, malicious systems could easily inject bad code into the system and thus attack both clients and services.

Jini provides for a complete architecture for discovery, configuration and access of distributed resources. Distributed collaborative service components could locate each other in quite a natural fashion using Jini. However, it requires each device to use Java technology for execution of bytecode. Services must be tightly integrated with the Jini architecture to benefit from its features. Besides, components like Web Servers, Lookup Services and RMI registries must be available in the network.

6.4.3.3 Universal Plug-and-Play (UPnP)

Another technology for device discovery is Universal Plug-and-Play (UPnP) [41]. Though being platform-independent, it emphasizes device configuration, location and access in a small network.

UPnP is based on a specific discovery networking architecture, with a simple discovery protocol sending announcements in specific HTTP messages over UDP and multicast. Clients can also discover devices by a specific discovery request over HTTP over unicast UDP. Besides, an event notification framework is defined to inform clients about state changes. Service access is tightly incorporated by the architecture, because clients can control devices and their services either via web interfaces or SOAP.

The use of UDP and multicast and the incorporation of service access is questionable concerning security. External security mechanisms can be included, but external certificates must be used, as long as IPSec is not generally implemented. Besides, the overhead for discovery and service access is not negligible, thus the approach is not scalable. Since the simple discovery protocol offers no filtering, UPnP is not applicable to larger networking environments where many services of the same type may reside.

6.4.3.4 Comparison

Many service discovery protocols have emerged and are used within their defined usage spectrum. It is difficult to decide which protocol is best suited for our use. Device discovery is a side aspect in our case, though in home networks users may want to auto-configure their environment. However, it will be decided by device manufacturers which protocol is supported. Similar aspects hold for service providers. Service location gateways can be a solution if different protocols exist in a certain environment [88].

Content and user location are further aspects that are only partly covered by existing approaches. UPnP together with the service registry system UDDI (Universal Description, Discover, and Integration [24]) can provide for a Content Directory in cooperation with web media servers. Similarly, user registers could be queried to locate user addresses. However, in our case we emphasize the location of content and users in a very restricted scope, like " getting the presentation of my friend in the same network". Thus, we use registrations with our system together with the SIP dialog event package to retrieve this information.

We choose SLP as the service location protocol, because SIP service templates are already available for this. Besides the concept is easily extensible to further services and does not require a specific programming environment or specific components to exist in the network. In the rest of this work we assume that client devices are either SLP capable or can be configured with addresses of proxy servers. If other discovery protocols are used by clients, our collaborative streaming architecture need not be completely redesigned, just the appropriate discovery modules have to be changed.

6.5 Summary

In this chapter, the most important Internet application-layer protocols to support streaming and conferencing systems are described in detail. The widely-used stateful Real Time Streaming Protocol (RTSP) is chosen for streaming session management in this work. After presenting a general centralized conferencing model, the Session Initiation Protocol (SIP), which is well known from its IP telephony support, is described in detail. Besides a number of other features, it offers support for centralized conferencing by the management of the SIP dialogs of a group within a so-called conferencing focus.

Both SIP and RTSP use other supportive protocols like the Real-Time Transport Protocol (RTP) for data transport and the Session Description Protocol (SDP) for description of media endpoints or presentations. Furthermore, other configuration approaches for multimedia services, comprising meta data or service location, has been described, though these approaches are not applied to a large extent until the time being.

In the next chapter, the architectural design of the proposed collaborative service using standard Internet protocols is presented.

7. System Architecture

In this chapter, the architecture of an actually implemented proof-of-concept collaborative streaming system using the concepts developed in the previous chapters and based on the standard IETF signaling protocols SIP and RTSP is presented. We give an overview of the use of the relevant protocol messages and show how additional information related to collaborative streaming can be exchanged.

This chapter is organized as follows: First, the general components of a costream system realized as an intermediate system are introduced. After that, the protocol message flow is shown for different use cases as described in section 2.4. The second half of the chapter is dedicated to the description of the component design in general, whereas the implementation will be shown in the next chapter.

7.1 The Costream Service as an Intermediate System

In the collaborative streaming system, the two protocols SIP and RTSP, with only slight enhancements, implement the signaling for the session transfer and the session control functions for clients. In these protocols, proxies as intermediate systems are often used for routing (in case of SIP) or for scalability reasons. The intermediate system in our architecture goes beyond usual proxy capabilities, because the costream service is equipped with application logic to control session transfer and shared session control. Nevertheless, the costream service can be understood as a kind of streaming proxy: it is a substitute for the streaming server which controls access to the presentation objects on the streaming server, according to the Proxy design pattern definition [44].

The overall costream system architecture shown in figure 7.1 consists of streaming servers in the global Internet, the collaborative intermediate system and the costream users. The costream intermediate system may include additional SIP and/or RTSP proxies. SIP proxies are used for call routing purposes and help clients to resolve domain names or to register several client devices. RTSP proxies are often used for caching or transcoding purposes. However, the general architecture of costream does not depend on the availability of these services.

In our architecture, the streaming servers are RTSP servers which are able to stream media from recorded presentations on file repositories, i. e. to packetize media data according to RTP guidelines and add the necessary RTP header data. They allow clients to access their service via an RTSP proxy which also realizes the session sharing functionality. Caching is not required in our architecture, but it can provide for a more efficient service. A SIP proxy can be used as configured by the network access provider; it is useful for call routing and device registration if other means like DNS are not available.

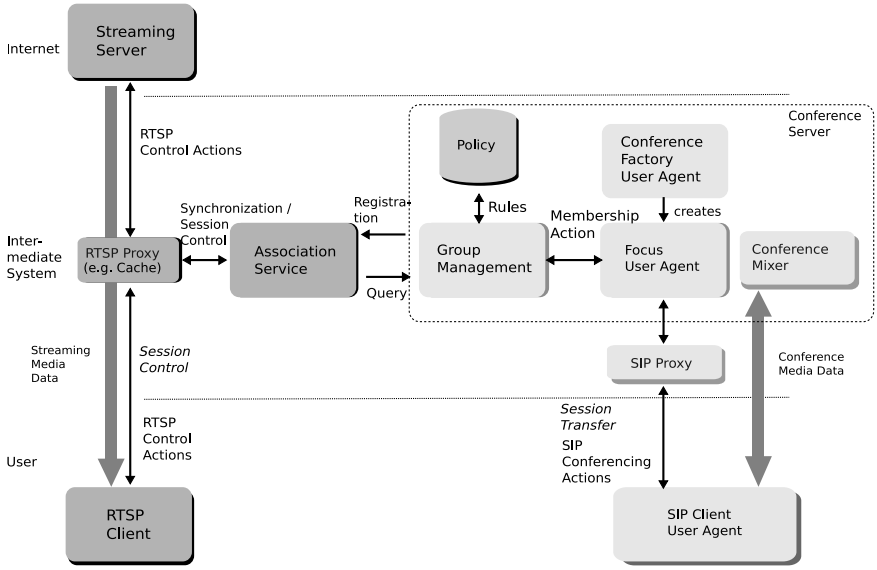


Figure 7.1: Signaling Architecture

The collaboration functionality itself is also divided into a SIP- and an RTSP-related part, depicted in the figure by the light and dark gray colored components, respectively. SIP methods are used for session transfer, and RTSP methods are used for session control. The *association service* implements the session sharing functionality presented in section 5.3.2, using different synchronization modules. The *group management* must control collaborative processing as shown in section 5.3.3. Its functions are called by a conferencing *focus User Agent*. The focus is created by a *conference factory User Agent*. The focus UA processes SIP conferencing methods that are forwarded by the SIP proxy and calls *membership actions* at the group management. The association service, in contrast, checks by communicating with the group management whether *session control actions* may be executed, and which reaction they produce. Additionally, the association service returns possible changes to session control requests to the RTSP proxy, which forwards these to the streaming server.

The costream architecture follows a centralized approach: all clients direct their signaling requests to one dedicated service. This approach is similar to the centralized conferencing frameworks proposed by the XCON WG [7] and by SIP [132]. We argue that this approach is reasonable, because the clients mostly are (logically) tightly coupled, since they want to synchronize their sessions, build associations within a group, and so on. If clients desire a looser coupling, they can use specific settings which prevent synchronization or notifications of group state. The costream service does not have to be implemented in one physical location, but the individual components could be distributed in the network. The group management service is logically coupled to the SIP focus UA, because the membership and group control actions are very similar. The conference factory UA, the focus UA and the group management can be implemented in a common *conference server* application. The term “server” may be misleading, but is often

used in SIP standards documents. This allows the use of standard SIP conference servers which only have to be enhanced by the group management module. Similarly, the association service can be integrated into a standard RTSP proxy and therefore be on a different machine than the group management service. Media distribution (depicted by thick gray lines in the figure) can be done completely separate from the group management functionality. Moreover,the costream architecture separates streaming media data transport from group communication media data transport. Conference mixers, which are standard RTP applications, are used to distribute media of a discussion channel to the group participants. These mixers do not have to process streaming media data at all, because these are sent on completely different paths from streaming servers via streaming proxies to clients.

A fully distributed approach would prevent a single point of failure, but is difficult to realize for general environments, because each client would have to implement parts of the above-mentioned focus functionality, in order to establish signaling relationships among group members. Additionally, the handling of a common group state with all the described facets would be very complex even for medium-size groups.

7.2 Message Flow for Initiation Transactions

Since communication among RTSP and SIP components must be set up, additional information that concerns the collaborative functionality must be given in the protocol requests. One design goal has been to limit changes in protocol requests to the minimum possible so that existing client implementations would require only few changes. Additional headers use the "X-header" notation so that no confusion with forthcoming standard RTSP or SIP extension headers can occur.

Before any initiation transaction can start, the clients and the conference factory UA have to register at a SIP registrar if calls have to be routed via proxies. Within the transactions, the created focus UAs are also registered. Consequently, clients do not have to call the actual machine the callee is running on, but can use aliases like `sip:conference.example.com` or `sip:group0@example.com`. The registrations are sent to the SIP proxy serving as registrar and are not shown in the following sequence diagrams for the sake of brevity. The REGISTER method of SIP is used as described in the standard and no modifications have to be made.

In the following subsections, we show sequence diagrams of the message flow among components of our costream architecture. Routing via proxies is omitted to keep the figures readable. We also show excerpts from the necessary protocol messages, depicting the most important header settings.

7.2.1 Start and StartGroup Transactions

The *Start* transaction is used to open a collaborative session for the initiator only. Other clients can join the session later. The message flow is shown in figure 7.2.

It can be divided conceptually into several parts, which are denoted by the numbers (1) – (5) in the figure. First, the initiator creates a group using the conference factory (1), which returns a concrete group URI of a newly created focus. The initiator uses this focus URI to join the group (2), before he/she registers the streaming presentation by sending a REFER message (3). In order to be informed about the state of the group, the initiator subscribes to the conference (4).

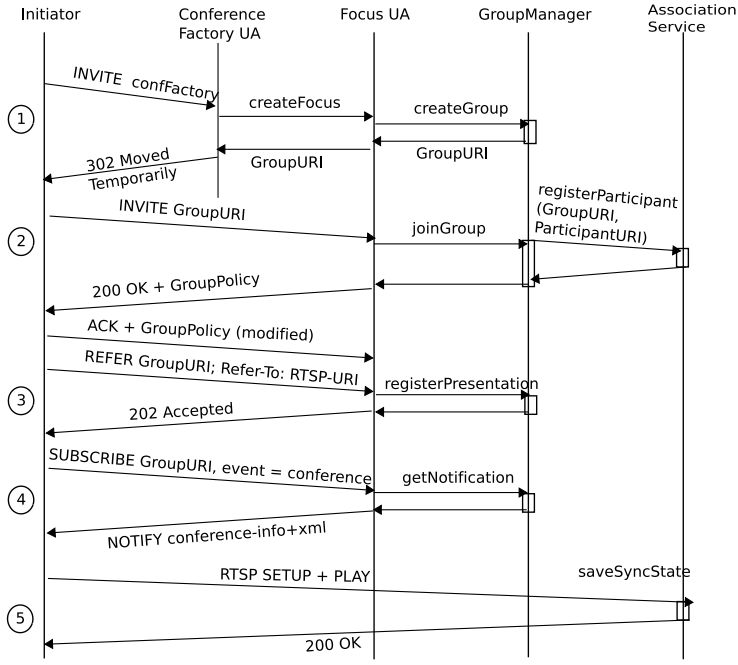


Figure 7.2: Message Flow for Start Initiation

Finally, the streaming session itself is set up using the RTSP messages SETUP and PLAY (5), shown as one operation in the figure for the sake of brevity. The association service must save the synchronization state of the initiator so that late-joining clients may play the presentation from the same play-time position.

The RTSP session setup in (5) can also be done before group creation or registration of the presentation. The association service must then assign users to collaborative groups. This will be delineated in section 7.4.2.1.

The individual parts of the transaction are described in further detail in the following subsections.

7.2.1.1 Creating the Group

The initiator creates a collaborative group by sending an INVITE request to the conference factory UA.

```
INVITE sip:conference@example.com SIP/2.0
From: "Alice" <sip:alice@example.com>;tag=1033
To: <sip:conference@example.com>
Contact: "Alice" <sip:alice@alicehost.example.com:5060;transport=udp>
Content-Length: 0
```

The factory creates a conferencing focus, which in turn creates a collaborative streaming group by calling the `createGroup` function of the GroupManager. This module checks if the caller is allowed to execute the operation. The policy for group creation must be defined separately from the group policies themselves. Further requests concerning the group are directed to the focus itself, which uses the returned group URI during the whole lifetime of the collaborative group. According to the proposal in the standard for call control in SIP conferences [77], the conference factory UA sends back a `302 Moved Temporarily` response, together with the contact information of the group.

```
SIP/2.0 302 Moved Temporarily
From: "Alice" <sip:alice@example.com>;tag=1033
To: <sip:conference@example.com>
Contact: "ExampleGroup" <sip:group0@conf.example.com:5060;transport=udp>
```

7.2.1.2 Joining the Group

The initiator now must join the group by sending another INVITE to the group URI, which causes the focus to request a joinGroup at the group manager module. The group manager also registers the group together with its initiator at the association service. Later joining clients direct their requests to the group URI itself and do not call the conference factory.

```
INVITE sip:group0@conf.example.com SIP/2.0
From: "Alice" <sip:alice@example.com>;tag=1033
To: "ExampleGroup" <sip:group0@conf.example.com:5060;transport=udp>
Contact: "Alice" <sip:alice@alicehost.example.com:5060;transport=udp>
Content-Length: 1800
Content-Type: multipart/mixed;boundary=xxx

--xxx
```

```
Content-Type: application/sdp
Content-Disposition: session

<Session Description not shown here>
--xxx
Content-Type: application/x-policy+xml
Content-Disposition: x-policy

<Policy Description not shown here>
--xxx--
```

Some extra information for collaborative streaming must be delivered for starting a group with the INVITE method. Since the group policy is negotiated at the time of group startup using a modified offer/answer model, either the initiating client or the focus UA have to deliver a policy offer. The initiator of the group sends the policy in the body of the INVITE message to the group URI. Since this body must contain a session description as described below, a *multipart/mixed* body must be used. Each body part can have its own content-type then.

If the initiator cannot define a policy for some reasons, the focus sends a policy offer in the body of the 200 OK response. It may make more sense to deliver a default policy offer in the redirection response of the conference factory, but the SIP standard requires that offers or answers are only delivered in reliable positive responses [135]. Redirection messages are not considered as reliable, because a proxy of the initiator's domain could use the redirection response to generate the new INVITE request to the given contact itself.

Until the time of writing, there is no standardized way to negotiate conference policies inside SIP. The XCON WG proposes to use a special Conference Control Protocol, but this would mean much more implementation effort. Of course, such a protocol can easily be used together with policy negotiation if it is available. Hence, we developed an own XML policy document format suitable for the needs of collaborative streaming. This document can be delivered as a SIP message body with the content-type *application/policy+xml*. The structure of this document is described in 7.4.1.2.

Additionally, a session description with content-type *application/sdp* has to be sent within the offer/answer model. This session description includes the media encodings and ports for the group discussion channel(s).

7.2.1.3 Registering a Streaming Presentation

In order to let later joining clients know about the collaborative streaming presentation, the RTSP URI of this presentation must be registered to the group management by sending a REFER request with the RTSP URI as the refer target. The group management distributes the information about the presentation URI using the conference event package, which is explained in more detail in the next section.

```
REFER sip:group0@example.com SIP/2.0
From: "Alice" <sip:alice@example.com>;tag=8331
To: <sip:group0@example.com>
Refer-To: <rtsp://example.com/movie.mp4>
Content-Length: 0
```

REFER also allows to add other resources accessible by URIs than SIP URIs. However, the standard does not define how to access these URIs. In costream, we add the received RTSP URI

of the presentation as a service URI to the group state and send a notification (cf. next section) to the subscribers of the group state. The clients transfer the RTSP URI contained in this notification to their RTSP client application before they set up the streaming session. It is thus possible to open a group first, add all members to the group, and register the streaming session afterwards.

For each received REFER request, the standard requires the creation of a subscription to the refer event package (see also section 6.2.2.6) so as to inform the sender of the REFER of the status of the resource that is to be accessed. The notification has to contain a fragmentary SIP status message. In costream, the creator of the group has to be informed whether registration of the streaming presentation has succeeded. Since the access of RTSP URIs is not standardized, a `SIP/2.0 200 OK` reply is sent in our architecture if the URL contains `rtsp://` and the collaborative streaming service is available, which also means that the association service must be connected and the group URI must be registered with the presentation URI. Enhancements could additionally deliver parts of the presentation description as part of the reply message. Otherwise, `SIP/2.0 503 Service Unavailable` is returned. The subscription to the refer event is terminated with the sending of this final notification.

7.2.1.4 Retrieving Conference Information

In every initiation transaction, the joining clients subscribe to the conference event package [137] using the SUBSCRIBE method, as shown exemplary in the following:

```
SUBSCRIBE sip:group0@example.com SIP/2.0
From: "Bob" <sip:bob@example.com>;tag=8037
To: <sip:group0@example.com>
Contact: "Bob" <sip:10.0.0.3:5060;transport=udp>
Expires: 3600
Accept: application/conference-info+xml
Event:  conference
Content-Length: 0
```

The notification server (not shown in figure 7.2), which is implemented on top of the focus UA, generates a notification by using the saved data from the group manager module.

The SIP standard conference event package delivers comprehensive information about the conference, its state and the participants. Since the XML messages of this event package can be long if all information is given, we use the possibility to give partial notifications instead of delivering the whole state, indicated by the use of the *state* attribute, which can take a value of *full, partial* or *deleted*. The notification for the first subscriber contains only the general conference description and the subscriber's own data. The conference event package also makes provisions for *roles* entries, which are filled with the respective entries from the group policy model. In addition to what is provided by standard conferences, the URL of the streaming presentation also contributes to the conference description as a *service URI*. The following example shows a notification with the most important entries. For the sake of brevity, additional information about the conference host and informational display texts have been left out.

```
NOTIFY sip:alice@example.com SIP/2.0
From: <sip:group0@example.com>;tag=6284
To: "Alice" <sip:alice@example.com>;tag=2519
Expires: 3600
Event:  conference
```

```
Content-Type: application/conference-info+xml
Subscription-State: active;expires=3600
Content-Length: 898

<?xml version="1.0" encoding="UTF-8"?>
<conference-info xmlns=".."
    entity="group0@example.com" state="full" version="0">
    <conference-description>
        <conf-uris>
            <entry>
            <uri>sip:group0@example.com</uri>
            </entry>
        </conf-uris>
        <service-uris>
            <entry>
            <uri>rtsp://example.com/movie.mp4</uri>
            <purpose>collaborative streaming</purpose>
        </service-uris>
    </conference-description>
    <conference-state>
        <user-count>1</user-count>
        <active>true</active>
        <locked>false</locked>
    </conference-state>
    <users state="full">
        <user entity="sip:alice@example.com" state="full">
            <roles>
                <entry>creator</entry>
            </roles>
        </user>
    </users>
</conference-info>
</xml>
```

Since associations are understood as conferences within a conference, *sidebar* tags are used to indicate associations inside a collaborative streaming session. Examples for this are shown in section 7.3.

7.2.1.5 Inviting a List of Clients

In the case of a *StartGroup* transaction, the initiator creates the group and joins it, using the same call control procedures as shown above. Afterwards, the messages shown in figure 7.3 must be exchanged. It is also possible to include a list of clients at the time of group creation, however it may still be desirable to add a group of clients to the existing group at a later point in time. The messages shown in the following figure can be sent at any time in the course of a presentation.

The group initiator sends a REFER request to the group management service, including the option-tag `multiple-refer` as defined by an Internet draft [19]. The list of clients is given in the body of the REFER request with the Content-Type of `application/resource-lists+xml` and the `Refer-To` header refers to the content ID of this list, which is indicated in a specific `Content-ID` header. Since the notification document cannot contain the state of multiple transactions to clients, the implicit subscription is suppressed by the `Refer-Sub: false` header and the option-tag `norefersub` as suggested by the Internet draft [19]. Instead, the initiator of the request is supposed to take the status information out of the conference event package.

```
REFER sip:group0@example.com SIP/2.0
From: "Alice" <sip:alice@example.com>;tag=8331
```

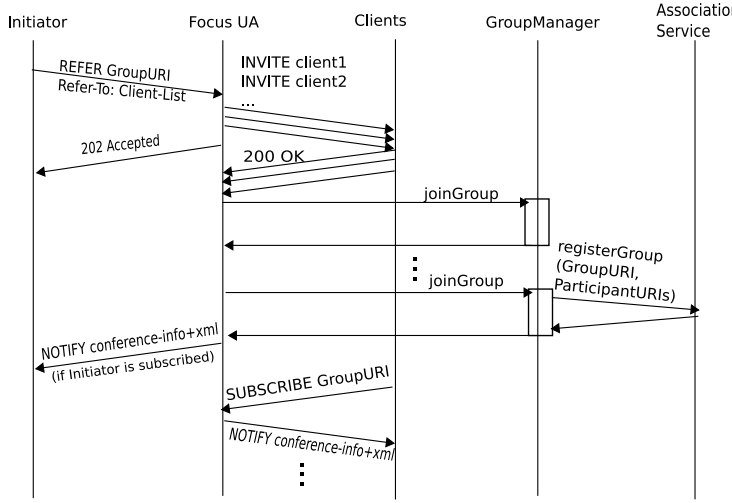


Figure 7.3: Message Flow for Start Group Initiation

```
To: <sip:group0@example.com>
Refer-To: <cid:myclientlist@example.com>
Refer-Sub: false
Require: multiple-refer, norefersub
Content-Type: application/resource-lists+xml
Content-Disposition: recipient-list
Content-Length: <content length>
Content-ID: <myclientlist@example.com>

<?xml version="1.0" encoding="UTF-8"?>
<resource-lists xmlns="urn:ietf:params:xml:ns:resource-lists"
           xmlns:xsi=...>
<list>
    <entry uri="sip:bob@example.com?method=INVITE" />
    <entry uri="sip:carol@example.org?method=INVITE" />
</list>
</resource-lists>
```

The focus then sends INVITE requests to all clients in the list. Those clients who answer this request positively are added to the group with one `joinGroup` call each. After all answers have been received, the resulting group members are registered at the association service. This late registration is more efficient because only one remote procedure call is required.

All clients subscribe to the conference event package to receive information on the presentation URI and on the other group members. For sake of conciseness, we show only one subscription in the figure. We have also left out the ACK methods of the INVITE three-way handshake.

7.2.1.6 Setting up the Streaming Presentation

Finally, the clients set up their streaming sessions using the RTSP SETUP and PLAY methods. Clients send the requests to the association service, which in turn initiates the synchronization

state at the first request and calculates the information for all following requests. The possibly modified requests are forwarded to the RTSP server, or – in case the reflected synchronization mode of the association service is used – to a reflector synchronization service.

Some extensions to the standard RTSP messages are needed to implement collaborative streaming. In RTSP, clients are identified using the session ID only, which is randomly chosen by the server. By registration of clients by the group management, the association server knows which clients belong to a certain group, but it does not know which session ID belongs to a certain member URI. To establish such a relation between RTSP session ID and SIP participant URI within the association service, a SET_PARAMETER message can be used. Every SET_PARAMETER request will be examined by the association service, which will set the contained parameter to the contained value if the parameter is known. Otherwise, the parameter will be forwarded to the streaming server.

```
SET_PARAMETER rtsp://example.com/presentation RTSP/2.0
CSeq: 421
Session: 12345678
Content-length: 29
Content-type: text/parameters

sip-uri: sip:user@example.com
```

To allow policy queries containing a certain participant URI, the SET_PARAMETER request has to be set before the member executes a control operation. Furthermore, the association service must determine to which association the streaming session has to be synchronized. The SET_PARAMETER request as shown above can also be used to find out the registered association ID. As an alternative to facilitate processing at the association service, an extension header "X-Association-ID" is introduced, which contains the SIP URI of the (sub-)group corresponding to the association. The presence of an association identification indicates that the shared state parameters of the (sub-)group shall be used for this streaming session. The association identifier also appears in subsequent control requests during the course of a presentation in case the client changes the association it belongs to.

```
SETUP rtsp://example.com/presentation RTSP/2.0
CSeq: 1
X-Association-ID: sip:group0@example.com
...
```

If the streaming session is set up before the collaborative group is defined, no association identification header is given. In that case, a new association is opened for the client and session state is saved. Later, the client can register the streaming session to a group and use its state to let other clients synchronize to it.

Standard streaming clients can also participate in collaborative streaming. However, for these clients it is not possible to set any parameters or additional headers. Therefore, all client and synchronization information must be looked up in the association module on the basis of the IP address of the client, which must be registered by the group management for that purpose. In this case, all clients must have different valid IP addresses. In a realistic environment, clients could also register to the collaborative streaming system using a login name, which would have to be set at both association service and group management. This information can then be provided by

the clients when they join a group and when they start a streaming session using authentication mechanisms. The association service uses the login name information for synchronization and for the policy query of the control operations. However, this approach would require additional registration and authentication procedures as well as a common registration data base.

7.2.1.7 Retrieving Streaming Session State

Information about the state of a streaming session must be deliverable to all clients in an association. This concerns the chosen set of tracks and, above all, the play-time position. This information can be used to print out informational text or to set time sliders within the (collaborative) client application. An extension that allows to communicate that information is especially useful if an association changes play-time state, because clients cannot be informed about the state change with standard RTSP. Instead of sending this information with the conference events, we have chosen to add an own event package which clients can explicitly subscribe to. Though this means more bandwidth consumption due to the exchange of subscription and notification messages, it is required so as not to disrupt conference event document parsers which would be unable to read the additional fields.

Thus, for each association within a collaborative group, the so-called *streamstate* event package delivers

- the association's SIP URI in the *uri* tag,
- the *playout-state* which can take values of "not started", "playing", and "paused",
- the *playtime-pos* which denotes the current play-time position that the association service has delivered to the group management, together with a timestamp,
- and a list of *tracks*, which in our case just contains the track identifiers (meta data may be added if descriptions provide for them).

The sidebar associations are contained within the *subgroup-streamstate* tags.

An example of a streamstate notification is shown in the following excerpt:

```
...
Content-Type: application/x-streamstate-info+xml
Subscription-State: active;expires=3600
...

<?xml version="1.0" encoding="UTF-8"?>
<streamstate-info xmlns=".."
  entity="rtsp://server.example.com/movie.mp4" state="full"
  version="0">
  <uri>sip:group0@example.com</uri>
  <playout-state>paused</playout-state>
  <playtime-pos timestamp="20071130T135040.30Z">125.0</playtime-pos>
  <tracks>
    <entry>trackID=3</entry>
    <entry>trackID=4</entry>
  </tracks>
  <subgroup-streamstates state="full">
    <entry>
      <uri>sip:sidebar1group0@example.com</uri>
      <playout-state>playing</playout-state>
      <playtime-pos timestamp="20071130T135040.30Z">50.0</playtime-pos>
      <tracks> ... </tracks>
    </entry>
```

```
  </subgroup-streamstates>
</streamstate-info>
</xml>
```

Clients subscribe to this package using a SUBSCRIBE request to the group URI with the event *streamstate*. The notification document generally includes subgroups. Thus, subscription to subgroups is not necessary. Since streamstate documents become large for groups with many associations, partial notification similar to the conference event package is used.

7.2.2 Copy and Move Push Transactions

Push transactions transfer streaming sessions from the initiator to a certain target. We assume here that group state already exists. The message flow for this case differs from the StartGroup initiation by the fact that the REFER is sent from one client to the other and the callee sends the INVITE itself, as shown in figure 7.4.

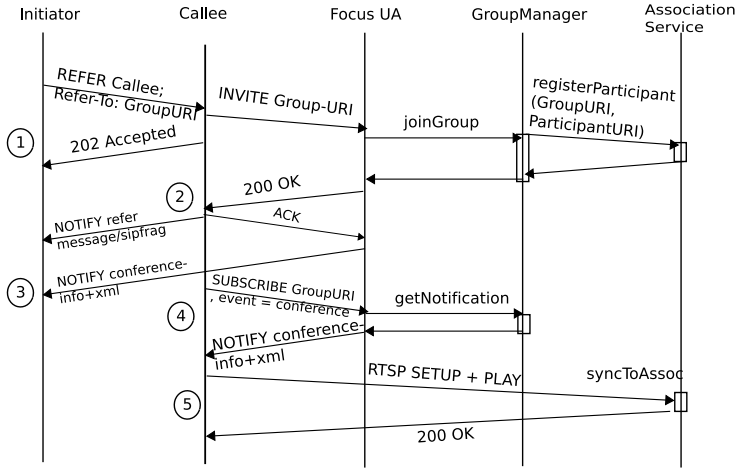


Figure 7.4: Message Flow for Copy Initiation

In the REFER method (1) , the `Refer-To` header contains the group URI. In this case, we assume that the callee accepts the invitation, but it may also deny it by sending a decline response (`603 Decline`). It is also possible to send a REFER to the conference focus with the URI of the user that should be invited, which causes the same processing as shown in section 7.2.1.5. This course of action can be advantageous in case the REFER method is not implemented by all clients or the focus must control who is allowed to invite others to a collaborative group. However, this message flow means more load at the conference focus, because the focus then has to generate notifications of the callee's state.

It is convenient for the callee if some information about the streaming presentation is given. In the push transaction, we use the `Subject` header together with the presentation URI for this purpose, because the conference deals with the presentation, and this header is probably displayed at any

user interface, thus users know that they are invited to a streaming presentation. If more meta data are available in future, the title of the presentation can be given in the Subject header, and a specific header can be used for the URI as an alternative. Note that this header is used for informational purposes, and is not required for the call flow itself.

As already mentioned, REFER methods create implicit subscriptions to the refer event. For this reason, the components that receive REFER requests must also notify the sender of these requests of the status of the action that is connected to the REFER. In this case, the procedure described in the REFER extension standard [151] can be used: For each REFER request that asks the recipient to send an INVITE to a certain resource, the final answers to that INVITE are given in the notification body as a `message/sipfrag` text. Thus, normally a `SIP/2.0 200 OK` will appear in the body if the invitation has been accepted (2).

As soon as the callee has joined the group, the initiator receives a notification about the newly joined member by the notification service of the focus (3). This is done using partial notification. Additionally, the callee subscribes to the group state information and receives a full conference information document (4). The callee uses the presentation URI from this document to set up its streaming session (5). The association service evaluates the `X-Association` header and synchronizes the play-time position to the existing group.

In case users do not want to create a tightly coupled collaborative streaming group and no synchronization has to be done, which is useful in spontaneous meeting scenarios, a client can use the REFER method without group interaction. We show this case in figure 7.5. The `Refer-To` header must contain the URL of the streaming presentation (1). If the callee client accepts the REFER, it can directly access the streaming presentation by requesting the description (2). The callee notifies the initiator about the successful access of the URI. It can then start its own streaming service by sending RTSP SETUP and PLAY requests. Note that clients do not have to direct their requests to the association service in this case. However, if they use the association service they will be able to define a group later and let other clients synchronize to it.

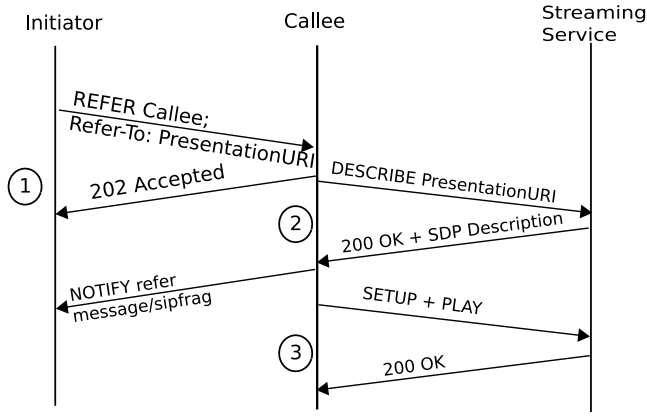


Figure 7.5: Message Flow for Copy Without Group State

The message flow in case of a move is similar to the copy case (see figure 7.6). Transferring

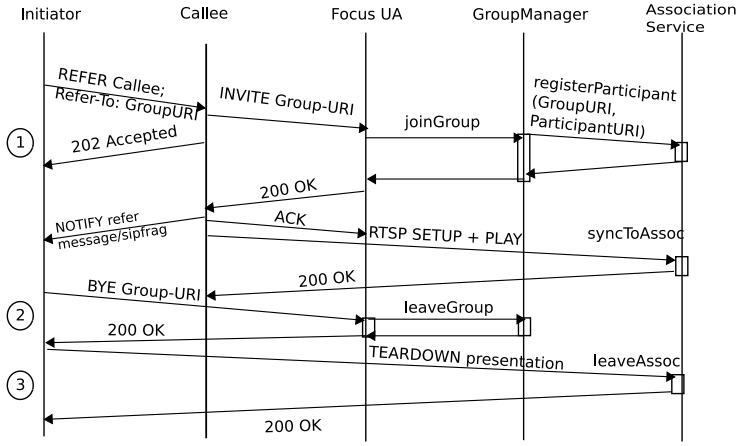


Figure 7.6: Message Flow for Move Initiation

the session to the callee is identical to the message flow for the copy (1). Subscription to the conference event package is not shown in the figure for the sake of brevity. After receiving the success notification of the refer event, the first client leaves the group with a BYE (2) and terminates its streaming session sending a TEARDOWN request (3).

7.2.3 Pull Transaction

In a Pull transaction, the initiator retrieves the streaming session from a participant as shown in figure 7.7. First, the Call-ID of the group must be determined. This is done by issuing a SUBSCRIBE request to the *dialog* package of any group member (1).

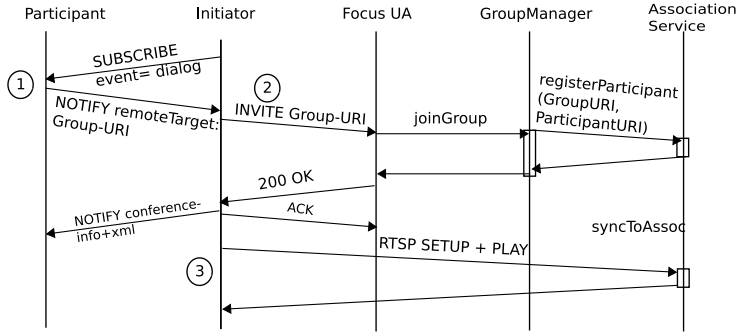


Figure 7.7: Message Flow for Pull Initiation

The subscription is used here with an expiration value of 0 so that only one notification about the current state of the user's collaborative sessions is sent.

```
SUBSCRIBE sip:alice@example.com SIP/2.0
From: "Bob" <sip:bob@example.com>;tag=8037
To: <sip:alice@example.com>
Contact: "Bob" <sip:10.0.0.3:5060;transport=udp>
Expires: 0
Accept: application/dialog-info+xml
Event:  dialog
Content-Length: 0
```

The standard for the dialog event package [138] only requires the state of dialogs to be sent inside notifications. This state is less interesting to us, because we assume that dialogs that reference collaborative streaming groups are always in `Confirmed` state. In order to keep notifications as short as possible, only the information of those dialogs is sent in the notification. In addition to the state, the notification delivers the dialog identification and the remote target's URI, which corresponds to the group URI.

```
NOTIFY sip:bob@example.com SIP/2.0
From: <sip:alice@example.com>;tag=4873
To: "Bob" <sip:bob@example.com>;tag=8915
Expires: 0
Event:  dialog
Content-Type: application/dialog-info+xml
Subscription-State: Terminated;reason=Timeout
Content-Length: 548

<?xml version="1.0" encoding="UTF-8"?>
<dialog-info xmlns="..." xmlns:xsi="..." xsi:schemaLocation="..."
      version="0" state="full" entity="alice@example.com">
<dialog id="451bf4847@alicehost.example.com"
      call-id="451bf4847@alicehost.example.com"
      local-tag="5209" remote-tag="7804">
<state>null</state>
<remote>
<identity display="...">sip:conference@example.com</identity>
<target uri="sip:group0@example.com"/>
</remote>
</dialog>
</dialog-info>
</xml>
```

The subscriber can use the group URI to join the collaborative group by using the same procedures as in the Push transaction message flow (corresponding to (2) and (3) in the figure).

A client that wants to pull a streaming session must know the address of one of the participants to be able to subscribe to the dialog information of this user. Such an address can be retrieved from SIP-enhanced telephone books. If a participant's address is not available it is also possible to subscribe to dialogs of a conferencing server, which can be discovered by SLP means. However, the conferencing server must implement more logic to deliver only the dialogs the client is interested in, and security and privacy problems may occur.

7.3 Message Flows for Update Transactions

In this section, the message exchange occurring if a client changes the play-time position within the presentation is described. Other changes like switching tracks (e. g. all members in a group switching to a track showing a certain camera angle) can be processed in a similar manner. The

control requests in this case would be a SETUP for adding a track and a TEARDOWN (with a specific track URI) for removing a track.

Since session control should be executed as done in common streaming applications, the client sends a PAUSE request or a PLAY request with a Range header for seeking to the association service. In standard RTSP, both pause and seeking are invoked by a PAUSE request. However, as the coming version 2.0 of the RTSP standard will allow immediate handling of PLAY requests in play state, the costream architecture already supports the distinction of *pause* and *changePlaytime* actions. The Darwin Streaming Server, which is used by our proof-of-concept implementation, also supports this.

The association service must query the group manager for a reaction (cf. section 5.2.2) on the control operation. It must include the SIP URI of the participant in this request. Hence, the participant must have sent a SET_PARAMETER request to create a relation between session ID and its SIP URI before any control request can be sent. This SET_PARAMETER request has already been used to allocate the participant to the correct association as shown in section 7.2.1.6.

The group manager sends back either the group URI if all members of the association have to change their play-time state or a newly generated sidebar URI if the association service opens a new association on behalf of the requester. If an action is not allowed, the group manager returns a negative response, which lets the association service deny the session control action. In this case, a negative response (`403 Forbidden`) to the RTSP control request is delivered to the client, but this client may still participate in the presentation.

The sequence of a group update with the result that all members change their state is shown in figure 7.8. In this example, the streaming presentation is paused by the initiator (1).

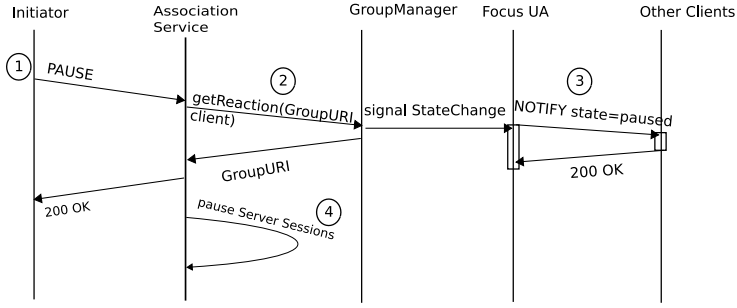


Figure 7.8: Message Flow for State Update for all Group Members

After the group manager has sent back the group URI (2), all group members are notified of the state change (3) with the *streamstate* event. The association service also pauses all server sessions of the association members (4), thus the clients need not send RTSP requests on their own. A disadvantage of this approach is that the RTSP clients themselves are still in play state, because no RTSP response is delivered there. However, the SIP client application may set the player state accordingly if the player can be controlled by keystrokes. Any additional pause requests from association members do not break the association: The association server does not query a reaction, but just sends an OK response. For the case of jump, the association server also forces

all server sessions to jump. Again, the SIP client application may give the new play-time position as an information to the media player so as to set a time slider to the correct value.

Alternatively, the clients themselves send a control request to the association service after they have been notified by the streamstate event. In that case, the association service must not query a policy decision from the group management, but re-synchronize the client to the changed state of the association. This form of updating the own session state only has to be done if the association service uses the independent synchronization mode, because otherwise the reflector will change the session state for all members of the association (cf. also section 7.4.2.3).

The message sequence for a group update where a new association is opened on a repositioning request is shown in figure 7.9. Again, the initiator sends a PAUSE request (1) and the association service asks for the reaction of the group manager (2).

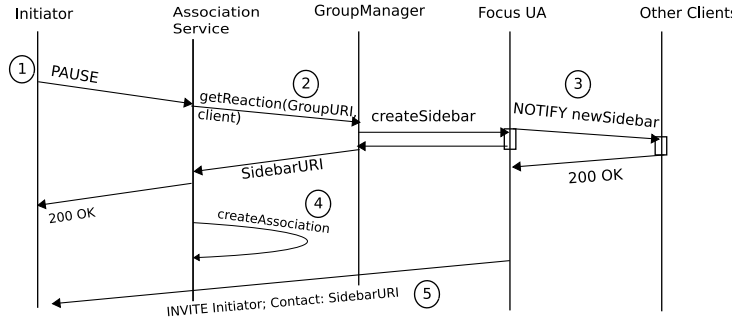


Figure 7.9: Message Flow for Opening of New Association

The group manager must invoke the opening of a new association on the focus, which opens a new sidebar to the conference and returns the sidebar URI to the association service. The other group members are informed by usual conference event notification if they have subscribed to this event package (3). Additionally, a streamstate event is distributed which informs about the play-time position of the new association. The association service creates a new association (4) and initiates it with the desired resume position (or the desired seek position). A re-INVITE sent to the initiator (5) is reasonable to update the remote target in the invite-initiated dialog, and to update media settings if a new group communication channel is opened.

Clients can be added to an existing association instead of opening a new association if an existing association has the required play-time state. In larger groups, this is reasonable for efficiency reasons, because less state must be kept.

Synchronization to an existing association is shown in figure 7.10. There is no specific request in RTSP for this synchronization, because group features are missing there. Thus, in our architecture the sidebar is joined by means of SIP, sending an INVITE request to the sidebar URI. The group manager registers the change at the association server, which synchronizes the member after receiving the following control request from the RTSP client. If the client includes the URI of the association it wants to synchronize to in the PLAY header, the processing is straightforward, otherwise the association service will have to lookup the correct association from the registration information which has been sent by the group manager.

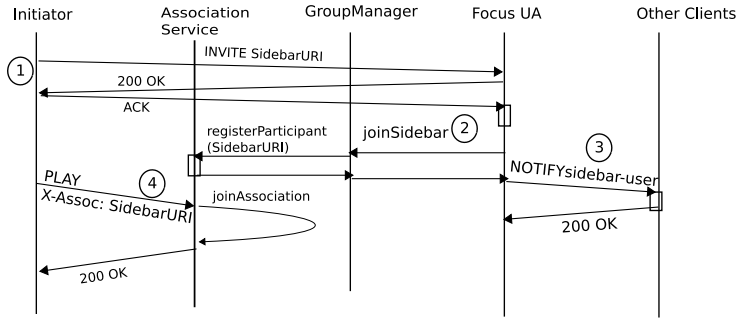


Figure 7.10: Message Flow for Synchronization to Association

7.4 The Costream Intermediate System

As mentioned before, the costream service must implement session sharing and group management services. We have designed the session sharing service as a back-end to a standard RTSP proxy and the group management as a back-end to a standard SIP conferencing focus.

In our first architectural design presented in [79] the Costream Proxy included both RTSP and SIP processing, mapping all requests into functions to manipulate a common data structure. While this is normally unproblematic in home environments, this solution will probably not work in other networking environments, so we decided to separate the synchronization from the group management. The advantage is that the services can be distributed on the network components in an easier way, and already existing conferencing and proxy components can be used and enhanced.

The group management and association services must communicate their state to each other. As we have shown in the foregoing sections, the group management must register groups and their members at the association service. It is assumed that members start streaming in a common association. Subsequent control requests that are sent to the association service by clients may split these associations. Since this must be done according to the group policy, the association service must either retrieve this policy from the group management or check each control request at the group management. We have decided to use the latter approach, because we expect only few control requests during a presentation. Additionally, notification of the other members about the new state is easier this way, because the group management can signal this state change to the notification server.

7.4.1 The Group Management Service

The group management service component must initiate and store the group state and notify the association service about it. We have already mentioned in section 5.3.3 that the group state consists of the association state and the group management state.

The architectural overview of the group management is shown in figure 7.11. A conference factory creates focus user agents on demand. In the SIP conferencing standard [132], one focus UA per conference is used. We adopt this design, even though the number of objects may be larger with this decision than with using a common focus UA which manages all conferences.

However, we use a 1:n mapping of the SIP stack to the focus UAs, i. e. all user agents use the same port, just the handling of SIP requests differs.

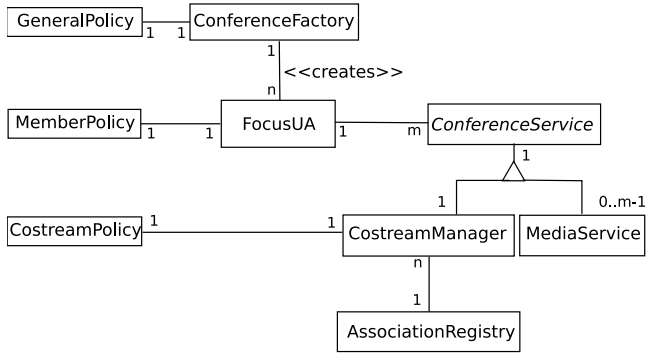


Figure 7.11: Conference and Collaborative Group Management

The group management application distinguishes requests as follows: If the Request-URI is the conference factory URI and the method is INVITE, another group is initiated; if the Request-URI is a focus URI, the request is intended for the focus; otherwise, the request may be for a sidebar (which depends on a focus).

The abstract type *ConferenceService* is the base class representing the services selected by the participants: A *CostreamManager* instance represents the streaming presentation, other conference services can be specific media services (e. g. a *MediaMixer*). Each focus has at most one CostreamManager, but may have several media services. The CostreamManager is instantiated at the time the streaming presentation is announced to the focus, which has been described in section 7.2.1.3. It has to register the participants of the group at the AssociationRegistry, which maintains the appropriate communication with the association service.

Different policy types control the handling of certain requests. The *GeneralPolicy* checks if a certain user is allowed to create a conference or collaborative streaming group. This policy must be configured at the conference factory UA by an administrator or specific control protocol. Join requests must be controlled by the *MemberPolicy* which is related to the FocusUA itself. The most important policy for collaborative streaming is the *CostreamPolicy* which is needed every time the association service gets a control request.

7.4.1.1 Group Relationships

A class diagram of parts of the group management in figure 7.12 shows the relationships between the modeled resources in our system.

Groups, participants and conference services like the costream manager are derived from the abstract type *Resource*, because they all have common attributes like URI and display name, and, more important, because they can be attributed to groups. Groups are *Composites* [44] which contain a number of participants, one costream manager, and possibly a number of subgroups. Subgroups are also instances of the type *Group*; they also contain a number of participants, but may not contain a presentation or other subgroups. We use a flag for the distinction of top-level

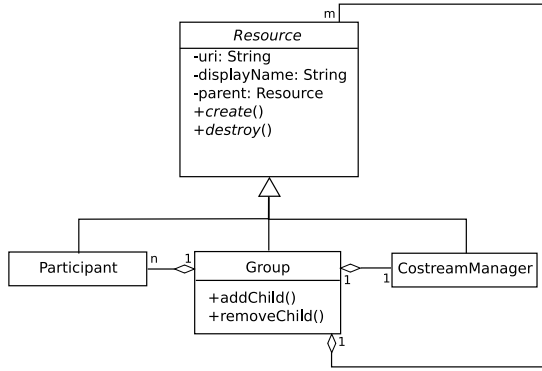


Figure 7.12: Composition of Groups with Resources

groups and subgroups. This has the advantage that a subgroup can easily be "upgraded" to become the top-level group if required (i. e. in the case that the only remaining member of the top-level group leaves).

7.4.1.2 Costream Policy Management

The policy document is an XML document, which is included in an INVITE method body. In this section, the structure of this document and the semantics of the individual elements are explained.

The initialization of the association state requires a commitment concerning shared session-specific variables. The content of the shared state is important for the control actions which must be checked by association service and group management in cooperation. If the play-time position is included in the shared state, each PAUSE request (or both PAUSE and PLAY according to the new standard) must be checked. If tracks are included, SETUP requests must be checked. We define an element `<shared>`, which lists the shared variables in `<entry>` element. We also define the values of *playtime* and *track* with the above-mentioned semantics and *none* for no shared state at all. If a document does not include a `<shared>` element, *playtime* is assumed as a default value.

The group management state applies a role-based access control model using member roles, lists of permissions which indicate the transactions that are allowed to be executed by certain roles, and the reaction policy. The XCON WG discusses certain roles in a conference and for certain floors of the conference [114]. Since the association of roles to their permissions should normally remain constant, we define an additional role for the costream service: A *Controller* is allowed to execute session control operations in a presentation, indicated by the permissions *Pause* and/or *ChangePlaytime*. If an additional role to distinguish between pausing and changing the playtime is required (which may be reasonable in learning scenarios), we propose to use a role of *Questioner*, which is restricted to pausing the presentation.

Since users can assume several roles in a conference at a time, the Controller or Questioner roles can just be added to the set of roles of a user. In the policy document, the `<roles>` element includes a number of `<role>` elements. Each role has a *name* attribute and contains a `<members>` element. The contents of the `<members>` element are lists of URIs, where wildcards are allowed,

e. g. to represent domains. Additionally, a number of permissions is given, which may be fixed or editable by administrators. However, permissions should be edited rarely to avoid confusing participants.

Another part of the document is the definition of actions and corresponding reactions of the group management. For this, a `<transaction>` element contains pairs of `<action>` and `<reaction>` elements. The action is in our case *Pause* or *ChangePlaytime*, the reaction can be chosen from *ChangeState* or *ChangeAssociation*.

Thus, the policy document is structured like the following excerpt:

```
<shared>
    <entry>playtime</entry>
</shared>
<roles>
    <role name="Controller">
        <members>
            <entry>alice@example.com</entry>
            <entry>*@somewhere.net</entry>
        </member>
        <permissions>
            <entry>Pause</entry>
            <entry>ChangePlaytime</entry>
        </permissions>
    </role>
</roles>
<transactions>
    <transaction>
        <action>Pause</action>
        <reaction>ChangeState</reaction>
    </transaction>
    <transaction>
        <action>ChangePlaytime</action>
        <reaction>ChangeAssociation</reaction>
    </transaction>
</transactions>
```

Since the policy document may be modified during offer/answer processing at group startup, the final policy is established after the party who had sent the offer has received the answer. The focus must initialize the policy objects that are associated with the group: a *MemberPolicy* for handling join requests, a *CostreamPolicy* for collaborative session control, and *Media Policies* which contain possible limitations on sending and receiving media. Examples of such media policies are the media formats a mixer supports or limitations on the number of supported clients for media services. The costream and member policies are integrated into one policy document. The XCON conference roles [114] can be used for the member policy, because each member can assume several roles and permissions of the member and costream policies are disjunct. Administrative actions are handled in the MemberPolicy, because they are initiated by SIP methods in our architecture. In case a specific conference control protocol is used, a separation of both policies is reasonable.

Each policy class has an operation *isAllowed* with the intended action and the initiator of the action as parameters and a boolean return value. The policy looks up the roles of the member and decides if the requested action is contained in the permissions of the role. To change policy information during the session, a re-INVITE with the new policy document is sent. Only an administrator should be allowed to do this.

If no policies are given, which may be reasonable in spontaneous scenarios, the group management assumes that every participant is allowed to execute every action and the default reaction is to change the association.

7.4.1.3 Membership Management

At the time of group initialization, the conference factory creates a new URI and new management state for the group. Before the initiator can create a new group and join it, the permission to do this must be determined. This must be a kind of "global" permission, because there is no group policy defined at that time. This is decided based on the registration of users to the collaborative streaming service. Either the creation of groups is allowed to all registered users, or a certain status of the user is queried.

Participant information is composed of the role assignment and the SIP dialog information. Each member must be assigned one or several roles, according to the given member and costream policies. The information about which roles apply for the participant is included into the conference notification document.

The dialog information is required if a dialog with a certain user has to be retrieved to support requests inside a dialog initiated by the focus. This is used for re-INVITEs which have to be sent by the focus UA if the association or the media settings change.

7.4.1.4 Notification Server

The notification service holds lists of subscribers for different events. It is coupled to a focus UA, i. e. it receives SUBSCRIBE requests and sends NOTIFY requests via the SIP stack of the focus. On reception of a SUBSCRIBE request, the notification server has to undertake the following operations according to the SIP event notification standard [130]:

1. Retrieve the event from the request and check whether this event is supported. If this is not the case, send a negative response (`489 Bad Event`).
2. In case the event is supported, check if there is already a subscription of the requester.
3. If a subscription exists, examine the `Expires` header of the SUBSCRIBE: If it has a value of zero, the requester wants to be unsubscribed, otherwise the requester asks for renewal.
4. If no subscription exists, generate a subscription and add it to the list of subscriptions for this event package.
5. Send an OK response and generate a full notification.

The notification server must store a version number for each subscription, because subsequent notifications for one subscriber must contain increasing version numbers.

The event packages differ in handling subscriptions, state changes, and notifications. For example, some packages support partial notifications, while others generate full notifications on each state change. Additionally, the rate of notifications may be governed by the event package.

The following event packages are supported by the notification server of the group management:

conference The conference event package [137] deals with documents of type *conference-info*. The package allows for partial notifications. The mentioned conference control functions like join, leave, addSidebar, etc. trigger state changes handled by this package.

A special case occurs for join, because the joining member should not receive both the full notification and the partial notification going to all subscribers. Hence, partial notifications must be sent to all subscribers except the joining member. The standard conference-info document is enhanced by collaborative streaming information in the following fashion: The URI of the streaming presentation is used as a `serviceUri` entry, with a `purpose` value of "collaborative streaming".

dialog For the dialog event package [138], only full notifications are issued in our system. We also do not use long-lasting subscriptions, the expire time is always set to 0. Thus state changes of dialogs do not trigger notifications.

refer The refer event package [151] is required for reference to other SIP user agents or to streaming presentations. For both cases, an initial notification is generated, and responses of the refer target to the issued protocol requests generate a final notification.

streamstate For distribution of streaming session state, the proprietary streamstate event package, which has been presented in section 7.2.1.7, is used. Since the streaming server cannot initiate play-time position changes using RTSP, clients belonging to the same association have to be informed this way. As already mentioned, only full notifications are sent.

Consequently, the notification server offers the functionality to generate full or partial notifications. The notification documents are built according to the rules of the respective event package. Partial notifications, if supported by the event package, are generated on each state change. Otherwise, full notifications are sent after a state change. Note that the notification documents have to be adapted for each subscriber, because the version number is defined on a per-subscription basis.

7.4.1.5 Communication with the Association Service

The group management and the association service must communicate, because the group management is controlled by the client's SIP conferencing actions, and the association service is controlled by the client's RTSP streaming control actions, which are independent of each other. To allow session sharing, the group management registers all groups and their members at the association service.

If a user issues control requests during the course of the presentation, the association service queries the user's permission to execute these actions. The group management can then open a sidebar to the conference if a new association has to be opened, or distribute a new streamstate event if only the play-time state of the association is to be changed.

The details of the communication among the services depend on the implementation. If distinct locations are used, either remote procedure calls (RPC) or a proprietary TCP-based protocol can be used. The remote interface at the association service provides for an operation `registerParticipant(GroupURI, ParticipantURI)` to register a member with a certain group, whereas the remote interface at the group management provides for an operation `getReaction (GroupURI, ParticipantURI, ControlMethod)`. The implementations of the interfaces transform RPCs or protocol requests into those server methods. They actually are accessed by the "clients" which have to set up the communication according to RPC standards or use the defined protocol.

7.4.2 The Association Service

The association service is a back-end to an RTSP proxy, which only has to work as a message relay. It consists of management modules for associations, synchronization modules, and a communication module to the group management service. The diagram overview of the association service is shown in figure 7.13.

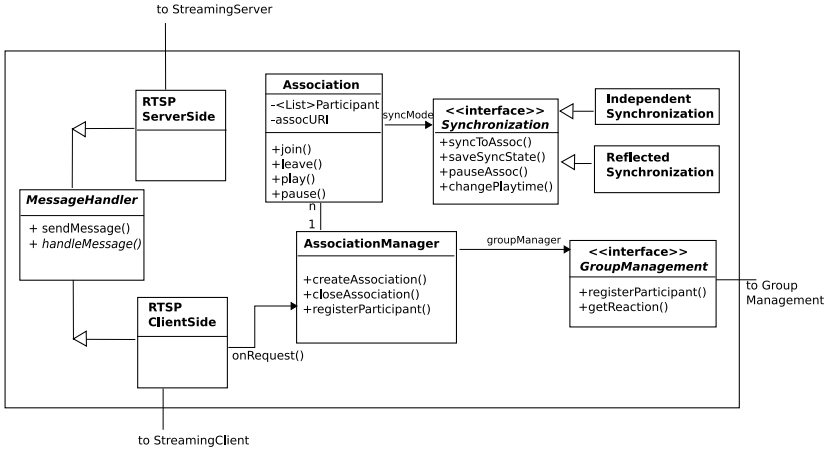


Figure 7.13: Association Service

The *AssociationManager* class is responsible for mapping RTSP protocol requests (operation `onRequest()`) to the corresponding association management functions. It maintains the list of associations and creates or retrieves associations. According to the protocol requests, operations of the classes representing these associations are invoked. There is no one-by-one mapping of RTSP protocol requests to association management functions, because these requests have to be handled differently dependent on the state of the collaborative streaming group and its associations (cf. also the more detailed description in the following section). The `createAssociation()` operation is called on the first SETUP request for a presentation if the specified association does not exist or none is specified. Additionally, new associations are created if the policy check reaction returned by the group manager requires this.

Associations do not keep references to other associations belonging to the same collaborative streaming group in our architecture, because the hierarchy of groups and subgroups is already maintained by the group management. Thus, associations act as containers for synchronization state and for their members. The association is in a certain state at any one time, which has effects on handling of join or control requests. This is described in section 7.4.2.2 in further detail. Another attribute of an association is the already-mentioned synchronization state, which consists of the play-time position in our case. The synchronization state is maintained by an instance of type *Synchronization*.

This interface type provides for four operations: `syncToAssoc()` is used to synchronize a certain user to an association, and `saveSyncState()` is used at the time the play-out starts in order to

save the state of the association. If the play-out is paused, the `pauseAssoc()` operation must set a resume position field to the current play-time position. If a PLAY is sent as a control request to change the play-time position, `changePlaytime()` is called to update the synchronization state. The different synchronization styles are, on the one hand, *IndependentSynchronization* which saves start time and start position to calculate the play-time position on demand, whereas the *ReflectedSynchronization* on the other hand maintains a shared server session with the correct play-time. Details concerning both synchronization modes are given in section 7.4.2.3.

An explicit module for communication with the group management service is used to register participants to certain groups or associations. This `registerParticipant` operation is called by the group management service, whereas the `getReaction` operation to check against policies is called by the AssociationManager for control requests. The communication between association server and group management is done according to the description in section 7.4.1.5.

7.4.2.1 Mapping of RTSP Requests to Association Management Operations

The mapping of RTSP requests to operations of the association manager is not trivial, because some RTSP requests are used for different functions within the course of a presentation. For example, PLAY is used at initial start-up as well as for control requests during the presentation. If a session is shared among a number of participants, the initial PLAY can also require to set the synchronization state or to retrieve the synchronization state, dependent on the number of participants.

The first SETUP request of each user is sent without a `Session` header. If the association's SIP URI is not contained or this association does not exist yet, a new association is created. If multiple tracks are set up, further SETUP requests are sent. These contain session IDs, so a Participant instance is created if it does not exist yet, and the session ID is saved.

PLAY requests have to be distinguished regarding whether they are used for initial session start-up or whether they denote a control request within a presentation. A flag within the participant information is used for this distinction. It is set after the first PLAY has been processed. The processing of the first PLAY is dependent on the state of the association: if the requester is the only member, the synchronization state must be saved, otherwise, the synchronization state must be retrieved. For further control requests, the distinction is similar: if the association has only one member, the control request can be permitted, otherwise, the group management has to be queried for the policy regarding this control request.

PAUSE requests are always control requests which are processed like PLAY control requests.

TEARDOWN requests are used either to end the whole session or to disable one track only. However, the latter is not supported by all streaming servers, e. g. the Darwin server closes the whole connection after a TEARDOWN. If the whole session is terminated, the leave operation is called on the respective association. Otherwise, two cases have to be distinguished: If tracks belong to shared state, the TEARDOWN to remove a track is a control request, which has to be handled like a playtime control request. If the selection of tracks is the individual choice of a participant, the request is forwarded to the server without further processing. The leave operation is also called in case a control request opens a new association. Thus, a participant is only member of one association (of a certain presentation) at a time.

The association manager must also handle requests of clients that cannot send association URIs in specific headers. An external media player may not be able to include headers with dynamic values. In this case, the participant's SIP URI is sent as a parameter.

In the following, the procedure to connect two clients that do not send association URIs to one association is described as an example. Assume that one client A starts a streaming session before starting a collaborative group at the conferencing focus. The AssociationManager thus opens an association and adds a participant A with its session ID, as shown in figure 7.14.

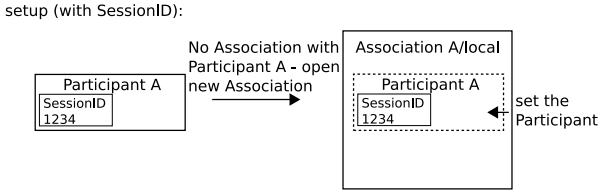


Figure 7.14: SETUP Processing

Client A must also send a SET_PARAMETER request including the participant's SIP URI in its body. This means that the association of A is retrieved by means of the Session ID and the SIP URI of participant A is then set. Figure 7.15 shows the processing of this request in the setParameter operation.

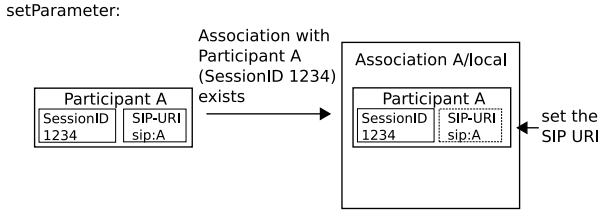


Figure 7.15: SET_PARAMETER Processing

At the time the collaborative group is started at the focus, the group URI is registered along with the participant(s). In the case shown in figure 7.16, two participants A and B are registered. The registration of A just sets the association URI (`sip:AssocA`) of the association, whereas a new participant B with its SIP URI is added at the second registration.

The streaming client of B then starts the session, again with a SETUP request (cf. figure 7.17). Since the AssociationManager does not know the relation of session ID and SIP URI, B also starts an own local association.

However, at the time when B sends both Session ID and SIP URI in the SET_PARAMETER request, the AssociationManager learns about this relation and can join B to the correct association. The local association of B can be closed, and both A and B belong to the same association, which is depicted in figure 7.18.

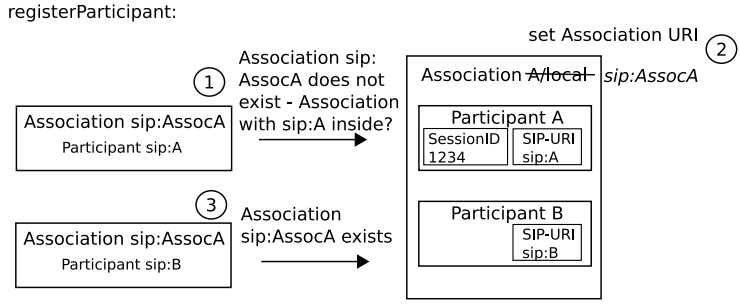


Figure 7.16: Registration of Participants

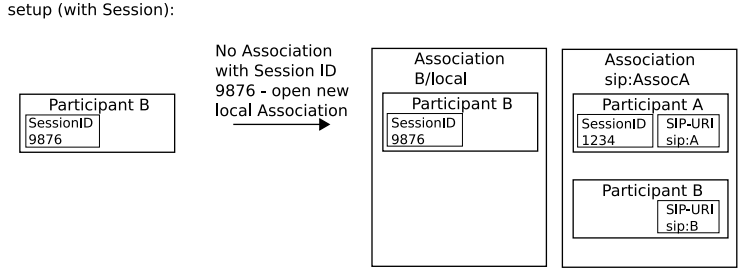


Figure 7.17: SETUP of Second Participant

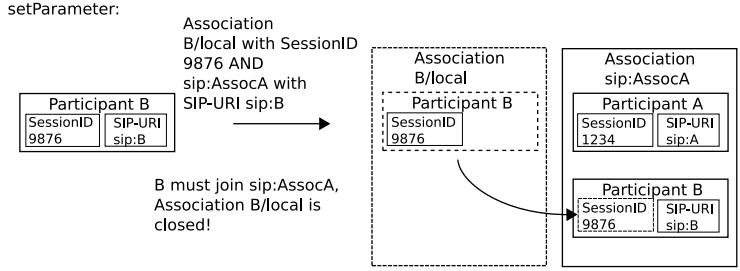


Figure 7.18: SET_PARAMETER of Second Participant

Hence, the AssociationManager must be capable to retrieve associations by their association URI, by the session ID of a participant, or by the SIP URI of a participant. It also must be able to transfer a participant from one association to another, and close the old association if necessary.

7.4.2.2 State of an Association

The state of an association during its lifetime is shown in figure 7.19. This state model is

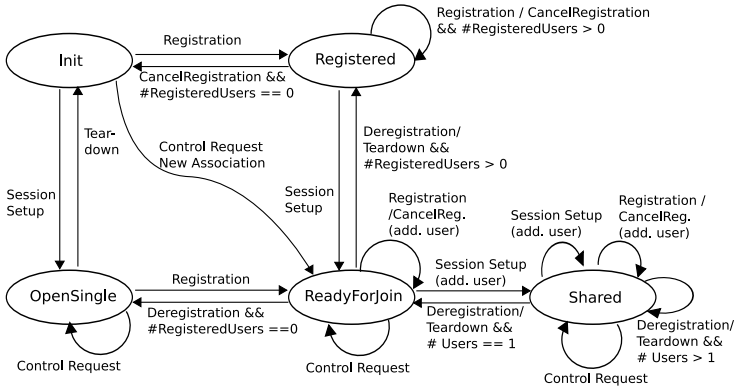


Figure 7.19: Association State

helpful for the AssociationManager to decide about the number of members of an association. Participant information can be added to the association using different requests, and the state of the participant itself does not have to be considered. For the sake of clarity, we have subsumed the SETUP, PLAY, and possible SET_PARAMETER requests into the notion of "Session Setup" in the figure. Note that a control request during presentation play-out is a PAUSE or a PLAY request for play-time position control, or a SETUP or a TEARDOWN request for track control.

The association is created in the *Init* state. A registration request coming from the group management transfers it to the *Registered* state, whereas the SETUP request leads to a transition to the *OpenSingle* state. From the *Registered* state, a SETUP leads to the *ReadyForJoin* state. This state can also be reached by executing the registration in the *OpenSingle* state. These different ways to reach the *ReadyForJoin* state are used because participants may be registered after session setup, in the case a user decides to open a streaming presentation before starting a group on the group management service.

In *OpenSingle* as well as in *ReadyForJoin*, only one participant is a member of the association. In this case, control requests need not be checked by the group management, but can just be forwarded to the server after saving the synchronization state.

In contrast to this, in the *Shared* state, which is reached at the session setup of a different user, the group management controls the access and returns a reaction. If the synchronization state is changed, the association will remain in the *Shared* state. Another possible reaction is that the requester itself opens a new association, which will be put into *ReadyForJoin* state, because the registration and setup has already been done for the client. The requester then leaves the old

association, which may be transfered into the *ReadyForJoin* state if the number of participants is equal to one after the control request execution.

The complementary action of a session setup is a tear-down. If the number of members in the association is greater than one, the association remains in the *Shared* state, otherwise it will return to *ReadyForJoin* for one remaining member or to *Registered* if no member is left. Even if the group members have stopped their streaming sessions, their SIP participant URIs persist with the association until the group or the members deregister their SIP URIs. The session ID in the participant information is removed to indicate a closed session. This is done to enable re-synchronization to an association in case the streaming session has broken down for some reason, for example during a handover in mobile environments.

There are two kinds of complementary actions to registration: A participant that is registered, but has not set up a streaming session (or has stopped it), may *cancel* the registration. In contrast, participants that have performed both registration and streaming session setup have to *deregister* from the association. Cancelling and deregistration are done by the group management using the same operation. The state transition that has to be performed has to be identified by reading the session ID from the participant information. Cancelling the registration means to remove a user from the association. The association remains in its state except in the case that no registered user exists, which means that the association is removed (transition to Init State).

For the user that is deregistered, the following assumptions are valid: An own association is built for the user and put into the *OpenSingleState* to allow a user to leave the group but keep the streaming session, which is useful in mobile scenarios. The user then may use the streaming session according to his/her own demands, e. g. execute control requests, or register an own collaborative group. The association without the deregistered user has to be considered also: If the association is in the *Shared* state and two or more users remain in the association, the association remains in this state. Otherwise, a transition from *Shared* to *ReadyForJoin* is made. In case that the association is in the *ReadyForJoin* state, the number of registered users has to be considered: if no other user is registered, the association state switches to the *OpenSingle* state (of course, no new association has to be built for the user in this case). Otherwise, a state transition to *Registered* is made, because there are still users that are registered and therefore interested to open a collaborative streaming session.

7.4.2.3 Synchronization

The association service is also responsible for saving and retrieving synchronization state. Thus, an interface is defined which provides for this functionality. The interface is implemented by different concrete synchronization modes, as already described in section 4.4.2.

Saving the synchronization state (`saveSyncState`) in the *independent* mode is done by saving start time and start position. If the operation `syncToAssoc` is called, this synchronization state is retrieved so that the current play-time position can be calculated to the difference of current time and start time, increased by the start position If an association is paused or the play-time position of the association is changed, the synchronization state is updated.

As an optimization, the play-time position is calculated like shown above, but the difference in the latency until the clients receive the first media packet is evaluated and added (cf. also section 4.4.2.3). This latency is approximated by adding the round-trip delay to the server (D_S) and the one-way delay to a particular client (D_{PC_x}). Delay measurements can either be

taken on application level (using signaling protocol headers or saving request sending times) or on transport or networking level (by measuring request-response delays of TCP or ICMP packets). Using application-level measurements has the advantage that server processing times are contained, which is preferable for the measurement of the round-trip delay D_S to the server. For measurements to clients, application level measurements are difficult to take, because the client would have to implement request processing. Hence, the round-trip delay to the clients is measured with a lower-layer method. The actual implementation of the delay measurements is discussed in section 9.2.3.3.

In the *reflected* mode, a reflector copies packets to each client in an association. Besides this initial synchronization, another requirement for a reflector to implement collaborative media streaming is the flexible reconfiguration of group streaming sessions, cf. also section 4.4.2.2. Since the association manager already decides when to change the whole association's play-time state and when to open a new association, the reflected synchronization mode can just execute a common method *changeServerSessionPlaytime* for control requests within an association. For new associations, the reflector is required to open a new server session. Clients that want to join an association are just added to the server session.

In our architecture, either an external reflector can be used, which means that client requests are forwarded to the reflector (tagged by directions on how the request must be handled), or an internal reflector has to manage client-server bindings on its own. In our example implementation, we use an external reflector as shown in figure 7.20, where the control requests of the client (1) are tagged by handling directions for the reflector (3) after querying the reaction of the group management (2). The reflector forwards the control requests (possibly modified according to the management of the reflector sessions) to the streaming server (4), which sends media data (5) which are handled by the reflector session appropriately (6).

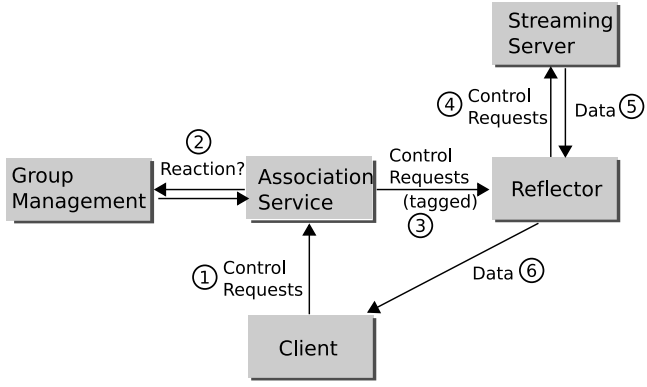


Figure 7.20: Reflector in the Costream Architecture

This reflector has been implemented as part of the *beaver* project [78]. Besides reflector functionality, this application implements caching to provide for a more efficient service. The existing reflector implementation groups clients just after their requested play-time, in order to serve requests from the cache if exactly the requested part of the presentation is available. Thus, in

a straight-forward implementation, the association service can calculate the play-time position just as done in the independent mode, and forward the requests to this reflector. Since the client destination addresses can be managed by the reflector, the association service does not have to be in the data path.

However, for a finer differentiation of groups and their associations, the synchronization module tags the requests by the operations the reflector should execute, i. e. `openNewServerSession` in the *saveSyncState* operation, `addToServerSession` in the *syncToAssoc* operation, and `changeServerSessionPlaytime` for both *pauseAssoc* and *changePlaytime* functions of the ReflectedSynchronization type. Consequently, we have added the processing of these tags to the Beaver reflector.

7.4.3 Discovery Components

As already mentioned, discovery in a collaborative streaming architecture means discovery of presentations, users, collaborative sessions, and service components. Presentations themselves can best be located by using content directories, though these are not used widely at the time being and will not be examined further in this work. The addresses of collaborative streaming users can also be retrieved using address directories or enhanced search engines. We have already shown in section 7.2.3 that collaborative session URIs can be located by subscribing to the *dialog* event of a user.

The Service Location Protocol (SLP) is used in the costream architecture to locate service components like SIP and RTSP intermediate systems. An SLP service template for the location of SIP proxies already exists [82]. The discovery of other costream components like the group management service (which is a specific SIP UA) and the association service (which is a specific RTSP proxy) follows similar principles.

The architecture of the discovery service used in this work is shown in figure 7.21.

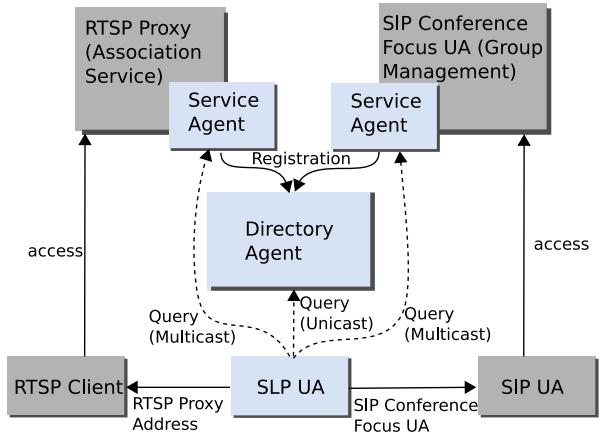


Figure 7.21: Discovery Architecture

Services like RTSP proxies (in our case, the association service) or SIP conference factory UAs (which host our group management) employ an SLP Service Agent. This Service Agent either

sends own service responses to service requests or registers with an (optional) SLP Directory Agent. In turn, clients use the SLP UA to query the services via multicast or via the Directory Agent. After the requested addresses have been retrieved, they can be accessed by the clients. The configuration can also be done in a static fashion so that clients need not implement any of the mentioned discovery components.

The service types that are used in our architecture are the already registered service types `service:sip:proxy` for an outbound proxy [82] and `service:sip:registrar` for a SIP registrar [83]. In order to discover RTSP proxies, we define the service type `service:rtsp:proxy`. Inside this service type, we define an attribute `costream-support`, which can have the values "association-service" or "reflector", to indicate whether the proxy implements an association service or a reflector. If an RTSP proxy service is advertised without this attribute, no costream support is assumed. For SIP conference factory UAs, we define `service:sip:conference`. Here, we define an attribute flag `costream-support` to indicate the costream support and we inherit other attributes from the abstract service type template definition. Thus, a user should always include the desired transport type (udp or tcp) in a service request as demanded by the SLP service template for SIP as an abstract service type [81].

The approach of locating SIP user agents by SLP could even be taken one step further: All user agents that are capable to participate in collaborative streaming could advertise this by an SLP Service Agent with an appropriate service type. This approach would not be applicable in the global Internet, but could be useful for locating users in closed environments like enterprise networks. However, we leave this case for future work and assume in our example implementation that user SIP addresses are kept in local address registers.

7.5 The Collaborative Streaming Client

The collaborative streaming client has been designed as a conferencing-aware SIP User Agent application. Additionally, it is possible to start an external media player, which includes an RTSP client. It is assumed that this media player can be controlled by keyboard commands. Therefore, a common graphical user interface with modules for both the SIP UA and the media player has been designed. SIP events are translated into the corresponding player commands (e. g. the client can use the notification of a play-time position change to update its own position). Available software components can be used in our design, however some slight modifications to protocol requests as already mentioned must be implemented.

We show a coarse overview of the costream client architecture in figure 7.22. Note that the MediaPlayerGUI and CostreamGUI boxes actually denote a number of classes which separate event and action processing from graphical display.

As already mentioned, the media player is an external program which can be controlled with keyboard commands and which has an RTSP client integrated. Since different such media players may exist or implementors may use integrated media players, we have separated the Media Player GUI from the Costream GUI. The Media Player GUI implements an interface which maps streaming events like the state change of users in an association to control commands of the media player. The Costream GUI generates these streaming events out of the set of costream events it receives by SIP interaction. Another task of the Costream GUI is to translate the user-initiated costream transactions into SIP requests. This is done according to the description in section 7.2. Additionally, events are used to gather information about group relevant state changes. Event

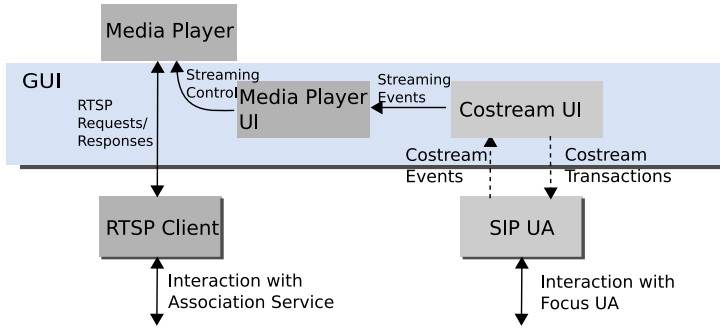


Figure 7.22: Overview of the costream Client Architecture

documents are delivered in XML in the SIP event notification system, thus the XML must be parsed and be translated into GUI actions. For example, the notification of a user's join operation is signaled by an event document that merely shows the information about this user. The user interfaces of other users should show this user as an additional member of the group. The user information can be shown on demand, too.

7.5.1 The Costream GUI

The costream GUI interacts with the human user of a costream client. If the costream client is a device which does not presume human interaction, a standard behavior must be defined for all events that require user interaction. We do not investigate further on this topic but assume that all these events, e. g. *ReceivedSession*, are accepted by default.

The GUI provides for operations corresponding to the initiation transactions. As shown in section 7.2, these transactions are translated into SIP methods (INVITE, REFER, or SUBSCRIBE), which are handed to the SIP UA. The SIP UA also processes received responses and hands related events to the GUI. Additionally, an operation to start a streaming session without initiating a group first is available at the Costream GUI. If this operation is executed, the media player GUI is initialized with the given presentation URI.

An overview of the events that are signaled to the GUI is given in table 7.1.

The *ReceivedSession* event is delivered if the client is invited to a collaborative streaming session. The URI of the caller and the presentation are shown to the client, so that the user can accept or deny the session. If the session is accepted, the media player is started with the presentation URI. In contrast, if another client wants to pull a session from this client, a *QueriedForSession* event will be handed to the CostreamGUI. Similarly to the handling of the ReceivedSession event, the user may accept or deny giving information about the conference dialog based on the caller information.

Two different events concerning notifications may be received: the *ReceivedNotificationMembership* event updates information on the group, based on the conference event package. In contrast, the *ReceivedNotificationPlayerState* event indicates a change of the streaming session state: The streaming session is paused or the playtime has been changed. This state change must be forwarded to the media player.

Event	Contained Data	User Interaction Required	Costream GUI Responsibility
ReceivedSession	Caller URI, Presentation URI	yes (accept)	Forward Presentation URI to Media Player GUI
QueriedForSession	Caller URI, Presentation URI	yes (accept)	Generate Notification on Dialog(s)
ReceivedNotification Membership	Conference Description	no	Update Display
ReceivedNotification PlayerState	Session State, Play-time Position	no	Forward Player State to Media Player GUI
ReceivedPolicy	CostreamPolicy	yes (modify, accept)	Update Policy Data and Display
ReceivedFailure	Failed Transaction, Reason	possible	Error Handling

Table 7.1: Costream Client Events

After an initiation transaction, the client is informed about the policy with a *ReceivedPolicy* event. If the client is the creator of the group or an administrator, the contained policy is displayed to the user, who may modify or accept it, otherwise, only the permissions are shown.

A *ReceivedFailure* event signals a failed transaction. The event contains the transaction and, if available, the reason for the failure. The user may have to react on the failure, or a certain error handling may be invoked.

7.5.2 The Media Player GUI

The media player GUI has been designed to assume control over the media player application. It provides for a simplified VCR-style user interface with play, pause and stop buttons. Additionally, a time slider, which can be dragged and dropped to desired positions, is available. Further control functionality can be implemented in future, but this interface suffices to support streaming client functionality in costream.

The media player GUI receives events which contain data that are relevant for streaming session setup and control. Particularly, on reception of a *ReceivedSession* event, the presentation URI must be forwarded to the media player GUI. The latter will then initiate the media player application with this presentation URI.

During presentation play-out, the user may control the streaming presentation if permitted to do so. Another possibility for control is the already mentioned event *ReceivedNotificiationPlayerState*, where a new play-time state of the client's association is described. This new play-time state must be translated into a pause or a jump. A pause keyboard command causes the media player application to send a new PAUSE request. Since RTSP allows sending PAUSE in the Ready state, the association service will answer this request on behalf of the server with a positive response, containing the resume position. The time slider is stopped at that position. A jump just sets the time slider to the play-time position indicated in the notification event.

7.5.3 The SIP User Agent

The SIP UA is a usual SIP User Agent client and server. It must be capable of sending and receiving the standard SIP requests. Additionally, the REFER extension must be supported,

which is required for the invitation of other SIP User Agents and for registering the streaming presentation.

For policy documents, multipart bodies have to be supported, because these are sent together with session descriptions during an offer/answer exchange.

Event notification has to be supported on client-side, i. e. SUBSCRIBE methods are sent, and NOTIFY requests are received. Notification documents of the *conference*, *refer*, and *dialog* packages have to be handled. Additionally, the client should support receiving subscriptions for the dialog event and generating dialog notification documents.

Clients that cannot support all mentioned extensions may still be invited to a collaborative streaming session. If they do not support REFER, the initiator of a push transaction may re-try the transaction by sending REFER to the focus, which in turn sends an INVITE to the callee. Their conference and streaming state may not be updated if notifications are not supported.

7.6 Summary and Discussion

In this chapter, we have shown the general architecture of a collaborative streaming system using RTSP and SIP. The most important design decisions of our architecture are summarized in the following.

A centralized architecture is used for group management and for play-out state management of a presentation, because most scenarios require a tight coupling of the participants. A distributed management of a common group state is more difficult to handle in terms of consistency. It is also expected that the target groups of the proposed scenarios have a limited number of participants. For this reason, scalability problems are unlikely to occur. Since the media distribution is handled separately from control processing, scalable media distribution methods can be applied if required.

The handling of SIP and RTSP requests is separated in the client as well as in the intermediate system. This is done to keep to the RFC standards as far as possible. Such a separation of conferencing and streaming functionality also enables the use of standard components. A SIP conferencing server (i. e. conferencing factory plus focus UA) enhanced with group management functionality and registration of streaming sessions and an RTSP proxy enhanced with association management and policy query features are sufficient to provide for a collaborative streaming service.

Clients have been equipped with full RTSP functionality. It would have been possible to “invite” the streaming server into the conference by choosing a multicast transport address where media data should be delivered to. However, a proxy would nonetheless be required to account for streaming control permissions. Additionally, some streaming servers do not support client-chosen destination addresses for security reasons.

Inside the association service, the management of associations has been defined. The so-called AssociationManager module must retrieve the correct association for each RTSP request. For this purpose, either a specific header or particular mechanisms that couple the RTSP session identifier to a participant’s SIP URI can be used. Additionally, the notion of internal state of an association has been introduced. This simplifies processing in the association manager and allows for temporal independence of registration and start of the streaming session.

We have defined an interface for the communication of association service and group management. The group management registers conferences and sidebar conferences as associations along with their participants, whereas the association service queries the group management for the group policy.

On the client side, the RTSP client must be configured with the participant's SIP URI. Additionally, notifications of play-time position changes may trigger changes in the RTSP client state. Hence, an interface between SIP User Agent and RTSP client has been defined.

An implementation of the architecture, consisting of the collaborative streaming client, a group management application and the association service, is shown in the next chapter.

8. Implementation

In this chapter, the actually existing implementation of the collaborative streaming architecture *costream* is described. It consists of three applications: the *Grappa* group management application, the *Cassis* association service, and the *Cream* collaborative streaming client. Grappa and Cream both are SIP User Agents, whereas Cassis is an implementation of an RTSP proxy. Cream also has an interface to an external RTSP media player application.

A Cream client starts a collaborative streaming session by using the Start or StartGroup transaction, which initiates an ad-hoc group on Grappa, with a number of other members for the StartGroup transaction. The Cream client also starts a streaming presentation using Cassis as its RTSP proxy. The start of the streaming session can be done before or after group initiation. Cream registers this presentation (either at the time of group initiation or explicitly after group start) at Grappa, whereupon Grappa registers the group with all its members at Cassis. Cassis then synchronizes all play (i. e. session start) requests for this group and requests access control from Grappa for following control requests.

Since parts of the applications are built upon external libraries, or use third-party components, those external parts are briefly described in the next section. In section 8.2, Cassis, an RTSP proxy and group synchronization application is presented. Thereafter, those classes which can be used by both Grappa and Cream are introduced in section 8.3. Grappa is distinct from Cream in that it has conference factory and focus features, and it may serve collaborative streaming groups. The implementation required to support these features is described in section 8.4. Finally, the Cream implementation is presented in section 8.5.

8.1 Third-Party Components and Libraries

Since it has not been a goal to re-implement all related protocols, we have used implementations of others where applicable. Particularly, an existing implementation of the SIP protocol stack has been used in order to be relieved of the task of parsing the variety of messages, headers, and parameters of this protocol. Hence, the NIST-SIP implementation has been used as the basis for this task. In case of RTSP, a comprehensive library which is yet open to extensions could not be found. For the association service prototype, we implemented RTSP handlers from scratch, because a number of messages and headers can be ignored for the session sharing functionality and just have to be forwarded. Moreover, the association service must process extension headers and parameter settings. Additionally, our architecture uses the reflector functionality of the proxy system developed in a parallel project called *beaver*. Besides, we used an external media player, which has been enhanced by an RTSP client implementation and which can be controlled out of our Cream implementation.

8.1.1 NIST-SIP

The implementation of SIP we have chosen for our project has been developed by the National Institute for Standards and Technology (NIST) [118] of the USA. It has been certified by the JAIN initiative [155], which aims to enhance and integrate telephony and computing applications to give people easier access to services. Thus, a standardized Java interface for SIP has been designed. It provides for *application interoperability*, since the underlying stack can be exchanged as long as applications only rely on the interface functionality.

The NIST reference implementation of JAIN SIP supports a variety of enhancement RFC standards, for example the event notification extension (RFC 3265 [130]) and the REFER method (RFC 3515 [151]). The API is described as a low-level API which exposes the SIP protocol layers to the programmer so that dialog-stateful, transaction-stateful or stateless applications can be designed (confer section 6.2.2 for a discussion of the different protocol layers). SIP User Agents, which are needed both for the Grappa and for the Cream application, mostly use dialog-stateful request handling, because the methods INVITE, REFER, and SUBSCRIBE all create dialogs if they are not sent within a dialog anyway.

In the following, the architecture of the JAIN SIP API is briefly described. An overview of how the single modules in this architecture are initiated at application start is given in figure 8.1.

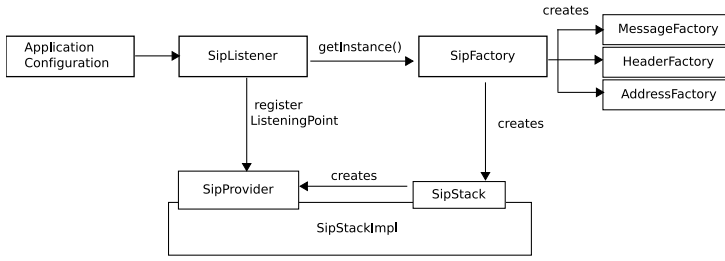


Figure 8.1: JAIN SIP Architecture [118]

Every application is required to implement the *SipListener* interface, which is used as the entry point for protocol stack configuration. A SipListener is notified of message *events* which contain incoming requests and responses. Each SipListener registers to a *SipProvider*, which manages a ListeningPoint and thus forms the interface to the underlying SIP stack. The SipProvider is also responsible for creating dialogs and client transactions. It provides for unique Call-ID headers to allow the identification of dialogs. Furthermore, requests and responses can be sent in a stateless way using the SipProvider. Note that the SipProvider and the SipListener have a 1:1 relationship. In the SIP stack itself, there may be one ListeningPoint for each transport protocol (at the time being, TCP and UDP).

The stack allows to set several property values. The setting of the stack name, identifying a specific implementation (in this case, *gov.nist*), is most important. Other properties provide for convenience functionality: The stack may be governed to handle all retransmissions automatically, to allow user-defined routing, or to use an outbound proxy.

JAIN-SIP creates all objects by *factories*. Four types of factories can be distinguished:

SipFactory creates the SipStack as shown in the figure, according to the specified stack properties. Additionally, the three factories mentioned in the following are created.

AddressFactory is responsible for the creation of SIP URI objects. Display names and URI parameters, for example the transport protocol, can be set.

HeaderFactory creates the headers of SIP messages. It provides a specific creation method for each header, dependent on the required parameter data. The headers are defined as sub-interfaces of a generic *Header* interface by JAIN SIP.

MessageFactory creates requests and responses. Headers like `To` and `From` are required at the time of creation, others can be added at a later time. It is also possible to parse a string into a request. Responses can be created based on a status code and the corresponding request, in this case they assume the headers of the received request.

JAIN-SIP provides for an easily accessible interface to SIP. It is low-level enough to give access to all header fields as required for implementing the User Agents and their back-ends, nonetheless it provides enough default values and convenience functionality to save implementation effort.

Unfortunately, the SIP proxy and registrar of the NIST does not handle the redirection sent by the conference factory in a correct fashion. Hence, no SIP proxy functionality has been used in the evaluation of the implementation. If such functionality is desired, the stand-alone proxy application of the reSIProcate project [129] can be used.

8.1.2 The Beaver RTSP Proxy

At the IBR, several research projects require a RTSP/RTP proxy implementation, for example the media data adaptation and collaborative streaming projects. Since no existing implementation could meet all the needs of the different projects, a new implementation called *Beaver* has been designed in a diploma thesis [96]. The requirements for this implementation were to provide for RTSP/RTP proxy functionality with several enhancements:

- a dynamic reflector to pass the same session data to several clients who join and leave the session in a dynamic fashion,
- a caching service to serve subsequent requests more efficiently, and
- a transcoding service for media data adaptation.

Besides, various signaling enhancements are required for initiating and managing collaborative and transcoding session state.

For collaborative streaming, the dynamic reflector functionality as described in section 4.4.2.2 is important. The proxy design separates control and data path, as shown in figure 8.2, with the RTSP control path classes in the upper half of the figure, and the classes which form the data path in the lower part.

The control path is again divided into client side (ClientRTSP) and server side (ServerRTSP), which both are derived from the RTSPMessageReceiver class and connected by the ContentManager class. On a client request, this ContentManager looks for a suitable server side interface, which can be a cached file in case another client has retrieved the presentation before, or a reflector session if another client is currently streaming the presentation at the correct play-time position. If no server side session is available, a new RTSP session to the server is built. This session can be cached or serve as a reflector session for future client requests.

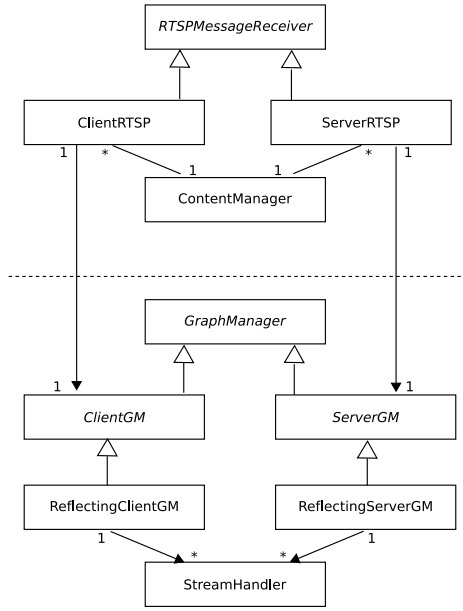


Figure 8.2: Beaver Proxy Design [96]

Beaver differs from other available reflector implementations in a feature essential for collaborative streaming: it allows to change the mapping between client side and server side sessions (cf. figure 4.7(b) also). Hence, clients are able to change the play-time position during presentation play-out, even if this means a reflector session has to be split. The ContentManager class will lookup a suitable server side session as described above. Of course, if a client is the only member of a reflector session, no splitting has to occur and the play-time state can be changed using RTSP means. Otherwise, if several clients reside in one reflector session and one of them requests a state change, the reflector either changes the play-time state as it is done for one client or splits the reflector session, opening a new reflector session for the client that sent the control request. The reflector decides on this according to the specifications of the association service.

Both client side and server side RTSP state machines follow the *State* pattern [44], with additional intermediate states *DescribeReq*, *ReadyReq*, *PlayReq*, and *TeardownReq* besides the usual states *Init*, *Ready*, and *Play*. The intermediate states indicate that a request is pending.

The abstract *Graph Manager* class is responsible for controlling data transport. Analogous to the RTSP layer, the Graph Manager is divided into client and server parts, which can be connected and disconnected in parallel to the corresponding RTSP layer classes. Concrete *ReflectingClientGM* and *ReflectingServerGM* classes are configured to set up stream handler paths as described above for dynamic reflection.

As already mentioned in section 3.2.1, the stream handlers are used to provide for flexible media processing. Each stream handler has one or several in- and output ports, which can be interconnected with other stream handlers in an arbitrary fashion. Since each stream handler is

responsible for one specific media processing stage only, changes affect only few parts of the implementation and new functionality can be added easily. Stream handlers can be active, passive, or hybrid. Active stream handlers initiate data transport and run in their own threads. In contrast, passive stream handlers only process data on demand of active stream handlers. Hybrid stream handlers work similarly but push their data downstream like active stream handlers do. For RTP, the ccRTP implementation of the GNU project [122], which is a C++ library providing for send and receive functionality of RTP packets, has been used. The active RTPSourceSH controls the reception of data packets from an RTP session, whereas the passive RTPSinkSH gets packets from a neighbor stream handler or the file system and sends them over the network using an RTP session.

A more detailed description of Beaver can be found in [96] and [14], together with the caching and transcoding implementations.

8.1.3 MPlayer RTSP Client

MPlayer [123] is a well-known open-source media player with support for a number of media formats. The MPlayer team has added an interface for the LIVE streaming library [93] to provide for RTSP streaming functionality. However, the LIVE library does not provide support for control operations. Thus, the komssys RTSP client implementation developed at the KOM institute at the TU Darmstadt, Germany [47], has been integrated into the mplayer. In future, the komssys implementation is to be replaced by the RTSP client implementation developed in the beaver project, but the evaluation in this thesis (cf. to section 9) is based on the komssys implementation.

The komssys RTSP client implementation provides for an *RtspPlayer* interface, which supports the well-known RTSP requests. Since mplayer does not support asynchronous response signaling, a blocking method to wait for the corresponding response is used. mplayer itself already provides an interface for handling data that are buffered from the network, thus only the corresponding buffer control structures had to be added, which give the mplayer code a pointer to the start, stop or seek functions.

Other parts of the interface concerning buffering of media data are very similar to the LIVE media interface or have been adopted from their code. For example, the frame rate must be calculated from the time-stamps of two RTP packets following in succession. Therefore, those packets must be saved to retrieve the necessary time-stamps.

8.1.4 OpenSLP

In our architecture, we use the OpenSLP [119] implementation for configuration of devices and services. OpenSLP has been implemented in C++ originally and is available as a source or binary package for a number of Linux distributions. OpenSLP contains an SLP daemon SLPd, which can act as a Service Agent or as a Directory Agent. Additionally, it contains SLP User Agent functionality inside a shared object library.

The client SLP UA multicasts a `SrvRqst` for the desired services. If these are registered at an SLPd Directory Agent this entity answers the query, otherwise the respective SLPd Service Agents give back the information.

A Java API, which is provided by the OpenSLP developer team, allows for easy integration with the Costream Client, Grappa Group management, and Association Service application classes. In

this API, the methods for registering and deregistering services at an SLP daemon are called in an *Advertiser* instance; both Grappa and Cassis use such an instance At the Costream Client side, a *Locator* instance, which finds service types, services, and attributes, is used. A *NetworkManager* class allows to connect the SLP Service and User Agents to the SLP daemon and to find SLP Directory Agents.

Since SLP service requests are normally sent using multicast, a multicast route has to be set. Since multicast is not globally available, we use this approach to lookup services in a local area network only. Extensions to SLP provide for remote service discovery by using DNS SRV lookups [175].

8.2 Association Service

The association service is implemented as an RTSP proxy which processes RTSP requests, and forwards them to the RTSP streaming server, possibly setting some header fields, for example the synchronized play-time position in the `Range` header. If executing some operation is not allowed due to group policy restrictions, it can generate a `403 Forbidden` response which is sent back to the client.

We have implemented an exemplary Association Service in Java. We did not put the Association Service into the data path in our test scenarios, so we left out RTP processing. For communication of the Association Service with clients and servers, we have used the Java Non-Blocking Sockets I/O package which is available since Java 2 Standard Edition version 1.4, because that way the necessity to use a separate thread for each incoming connection can be avoided.

8.2.1 RTSP Communication

An overview of the classes that handle RTSP requests and responses is shown in figure 8.3. This

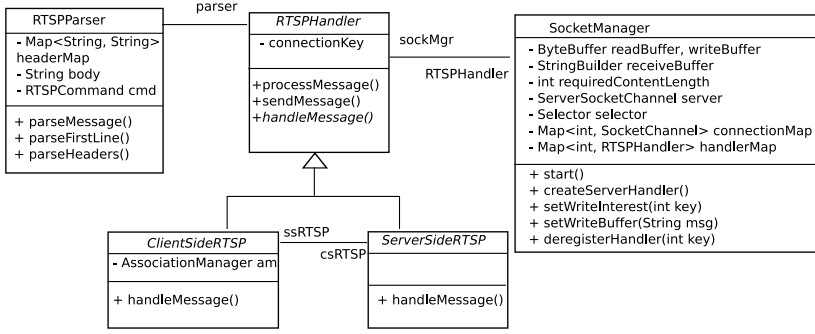


Figure 8.3: RTSP Message Handling

part of the implementation corresponds to the RTSP implementation of beaver [96] (cf. figure 8.2 also). However, a fixed connection of client and server side is used and no ContentManager class is required.

As an entry point to the RTSP communication subsystem, the *SocketManager* class handles incoming communication requests. It maintains read and write operations on the established socket channels and stores the appropriate socket channels and *RTSPHandler* object for each connection key. Besides usual read and write buffers, a receive buffer is used to store RTSP messages that are longer than the received packet from the transport protocol. The content length header is parsed by the SocketManager to decide when a RTSP message is actually complete.

The SocketManager also creates the *ClientSideRTSP* and *ServerSideRTSP* objects. Both classes are derived from the RTSPHandler class, since message sending and parsing are equivalent for both client and server side. Only the actual method processing differs. Thus, the RTSPHandler parses the message and delegates message handling to the ClientSideRTSP or ServerSideRTSP object. In case of the ClientSideRTSP, the `process()` method of each command (see below) is called, which may or may not call the AssociationManager. Incoming responses are handled by the ServerSideRTSP, which normally forwards them directly to the sending method of the corresponding ClientSideRTSP object. In order to send messages, the RTSPHandler must save a reference to the SocketManager so as to write the data into a write buffer and initiate sending on the socket channel.

The *RTSPParser* class creates method objects (*SetupCommand, PlayCommand,...*) derived from an abstract *RTSPCommand* class for every incoming RTSP command. All headers are parsed into a simple header map, and each method object stores information required for processing the respective RTSPCommand (like session ID or play-time position). The remaining headers are not needed by the association service, but are forwarded to the server or client, respectively. Each subclass of RTSPCommand class implements the abstract `process()` method, which calls the appropriate method of the AssociationManager class, or just returns without processing.

8.2.2 AssociationManager

The AssociationManager holds references to objects of type *Association* inside a map accessible by the association (SIP) URI as a key. Figure 8.4 shows an overview of the interfaces of both classes. Note that common setter and getter methods are not shown for sake of clarity.

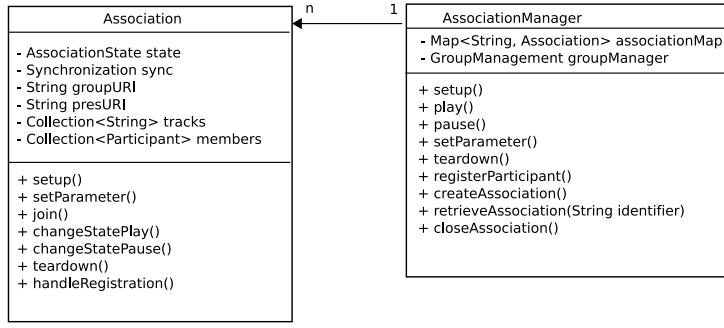


Figure 8.4: Association Manager

The AssociationManager implements the standard methods (`setup()`, `play()`, ...) called by the respective RTSP commands, and a `registerParticipant()` method which is called by the interface to the group management. For each method, it is the AssociationManager's responsibility to look up the correct association of the requester. This may include requesting a policy decision from the group management. Of course, creating and closing associations is another responsibility of the AssociationManager.

For clients which are not able to send association URIs in specific `X-Association-ID` headers, additional retrieval methods have been implemented, which give back an Association object looked up from the session ID or SIP URI of a participant. For this, a *Participant* class is used which holds both session ID and SIP URI of the client. To retrieve the correct association, the client must be able to send a `SET_PARAMETER` request containing a `Session` header with a session ID and a parameter body containing the participant's SIP URI (cf. the exemplary description in section 7.4.2.1). It would be possible to use a login name / password authentication, but this functionality has not been implemented yet, because it requires more message overhead. Before the `SET_PARAMETER` request is received, the AssociationManager builds a local association to keep the client's RTSP settings.

8.2.3 Association

The responsibility of the Association class is to provide methods for saving and changing its synchronization state, and methods for adding and removing participants. As already mentioned in section 7.4.2.2, associations are in a particular *internal* state at each point in time. This state should not be confused with the play-time state, which is visible externally. The internal state serves as a way to simplify message processing. It is implemented following the *State* design pattern [44].

Externally, a method exists for each RTSP request and for participant registration. A distinction is made between the first play request of a user, which is used to join an association (method `join()`), and further play requests, which are control requests within a presentation, and call `changeStatePlay()`. The internal state of an Association object can also be retrieved by the AssociationManager. This may be seen as a violation of design principles, but is necessary to distinguish the *shared* state from other states, because the policy of the group must be queried if a control request is received and several members share an association. In states other than shared the policy query is not necessary.

At the creation time of the association object, a synchronization mode must be set. The *Synchronization* interface provides for four methods as shown in section 7.4.2.3. These methods are implemented by the *IndependentSync* and *ReflectedSync* classes. For convenience, the play-time state can be queried, which is useful for testing purposes and to set the play-out state as a result of the participant registration method described in the next subsection. As already mentioned, the IndependentSync calculates the correct play-time position for a client and inserts this into the `Range` header. The ReflectedSync forwards all requests upstream to the reflector, but tags control requests to direct the reflector's behavior. The proprietary header `X-Reflector-Tag: <reflectorTag>` is used for this. The ReflectedSync methods insert the correct reflector tags into the RTSP message. We show a summary of the synchronization interface implementation in table 8.1.

For the time being, the choice of the synchronization mode has been implemented in a static fashion at the start of the Cassis application. Enhancements to select the synchronization mode for

Synchronization Method	IndependentSync	ReflectedSync
saveSyncState()	Set start time + position, calculate virtual play time	Set tag openNewServerSession
syncToAssoc()	Calculate play-time (corrected)	Set tag addToServerSession + sessionID
pauseAssoc()	Set resume position	Set tag changeServerSessionPlaytime
changePlaytime()	Set start time + position	Set tag changeServerSessionPlaytime

Table 8.1: Synchronization Interface Implementations

a particular association could use a flag sent by the group management at the time of registration. However, this would mean that sessions could not be set up before group registration, because it is hardly possible to move a session that has been set up without reflector usage to a session using a reflector and vice versa.

8.2.4 Interface to the Group Management

In order to receive registrations from the group management and to query the policy of the group management, a *GroupManagement* instance comprises the two methods `registerParticipant (associationURI, participantURI)` and `getReaction(associationURI,participantURI, controlMethod)`. This instance also contains a reference to the class actually implementing the connection to Grappa and a reference to the AssociationManager. The GroupManagement has only pass-through functionality, but has been designed like this in order to be able to replace the connection method to Grappa. Hence, Grappa calls the `registerParticipant` method which will register participants at the AssociationManager. In turn, the AssociationManager calls the `getReaction` method to query a policy decision of Grappa via the connection to it.

We have implemented the connection of Cassis and Grappa using Remote Method Invocation (RMI). Cassis is the server for the registerParticipant method and the client of the getReaction method, and Grappa just vice versa. Thus, both interfaces must be offered to the respective counterpart. We have chosen two interfaces CassisRemote and GrappaRemote for this. Each interface implements its server method, plus its own dedicated lookup name. The classes implementing the interfaces must then create a server instance and bind this to the lookup name using the RMI `Naming.rebind` method. The client method executes a `Naming.lookup` method and calls the method of the remote server interface.

In figure 8.5, an overview of how the methods of the different applications are called is shown.

8.3 Shared Classes

Since both conferencing focus and costream client are SIP User Agents, a number of classes can be shared among both applications. The notion of groups, participants and policies exists in both applications, thus the management of these entities can also be shared to a certain extent. A notification server also exists in both applications, though different event packages are supported. Hence, we provide for abstractions as far as possible and encapsulate differences in the respective implementation classes where necessary.

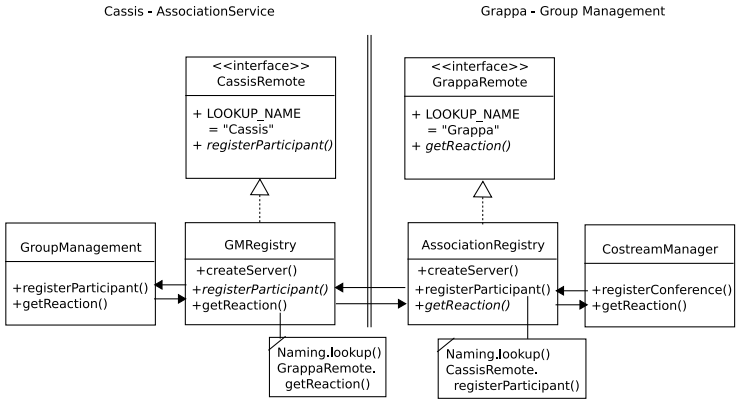


Figure 8.5: Remote Interfaces in Cassis and Grappa Application

8.3.1 SIP Stack Interface

A common interface package to the JAIN SIP stack (cf. also section 8.1.1) consists of a number of classes as shown in the following class diagram in figure 8.6.

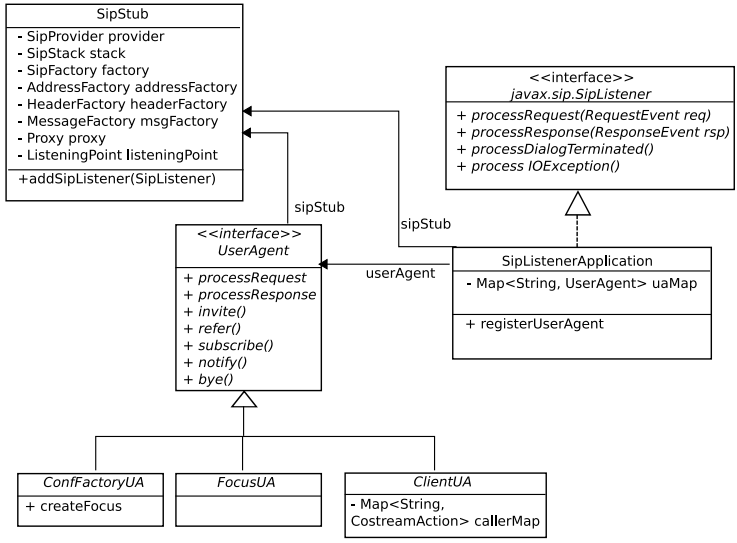


Figure 8.6: SIP Listener and User Agent Classes

SipStub is a singleton class which holds references to all objects provided by the NIST SIP implementation: The SipFactory is responsible for creating the SipStack as well as the address, message, and header factories. The SipStack in turn creates the ListeningPoint and the SipProvider

objects according to the configured preferences (see below also). A proxy can be used as a registrar and for call routing. The SipStub also registers the *SipListenerApplication* as an implementation of the *SipListener* interface to the provider. The SipListenerApplication manages the registration of concrete classes implementing the *UserAgent* interface. We have included the SipListenerApplication as an intermediate class to support a number of focus user agents within one application. For a mere costream client, the UserAgent itself could also implement the SipListener interface. The UserAgent interface abstracts from implementation differences of *ConfFactoryUA*, *FocusUA*, and *ClientUA*. Hence, the concrete implementations reside in the Grappa and Cream implementation packages and will be described in further detail in sections 8.4.1 and 8.5.1, respectively.

8.3.2 Configuration

For configuration, a *SipConfiguration* class is used to set properties like (among others) stack IP address, port, user name, and proxy address. It manages an XML configuration file, which has to be loaded, modified and saved on demand. A *ConfigurationParser* is used to parse the mentioned properties from this configuration file. The parser is based on *dom4j* [31], which has been used because of its support of XPath expressions [23], making the parser code easily understandable and extensible. If the user does not give an XML configuration file, default settings are used. In each case, the configuration can be modified in a graphical user interface using the *Preferences* component of the JFace user interface toolkit.

In both Grappa and Cream, specializations of the SipConfiguration class are used. They include configuration settings that are only relevant for the respective application. Moreover, the specialized configuration classes store references to global objects within the application. The configuration object itself can then be passed to the constructor of a class that requires access to these objects. For simplicity, one configuration object is used for the whole system. The *GrappaConfiguration* class extends SipConfiguration by references to the GUI, the SipListenerApplication, or the ResourceList objects. Additionally, the service configuration for the SLP discovery must be set. We define the constant *slpAbstractServiceType* with the value set to “sip”, and the constant *slpConcreteServiceType* with the value set to “conference”. We use only the attribute *costream-support=true*. In a real conferencing application, other attributes could be used to indicate support of certain media types and encodings.

In the *ClientConfiguration* class, there is a streaming client property which is the name of a class which has to implement the *CostreamApplication* interface as described above. In our case this name is "ExternStreamingClient".

8.3.3 Notification Service

A number of *Notification Service* related classes are used in both Grappa and costream client. The class diagram in figure 8.7 provides for an overview of them.

The *NotificationService* provides interface methods for managing subscriptions and handling state changes. It contains a reference to the UserAgent which is responsible for sending notifications over the SIP stack. Additionally, the *EventPackage* instances are contained for all supported event packages, which may be pre-configured or added on demand by the application. The *EventPackage* abstract class holds a map of subscriber dialog identifiers to their *Subscription* instances. It implements a subscribe and an unsubscribe method used by all concrete event

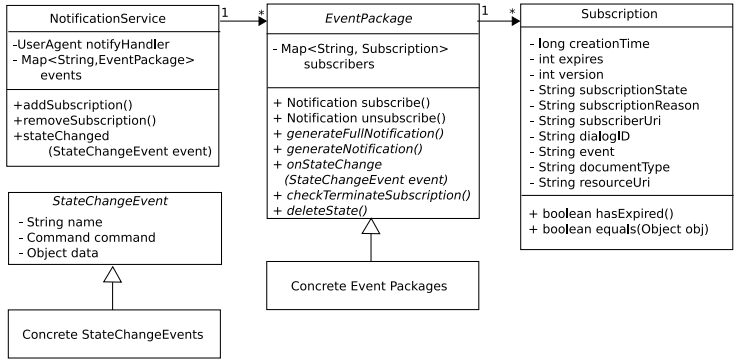


Figure 8.7: Notification Service

packages. Both methods operate on the subscriber map and return a full notification, which is generated by the concrete event packages. Those concrete event packages have to implement a number of abstract methods defined by the EventPackage class: *generateFullNotification()* collects the whole data set of the event package, whereas *generateNotification()* is called by the *onStateChange()* method and composes a notification of data which has changed only. The notification service must also decide on a state change whether it should terminate a subscription (for example if the resource does not exist anymore), and whether any data should be deleted, which must be done after notification generation. The *Notification* document (not shown in the figure) contains content that is ready to be sent as the body of a SIP method. Besides the actual text string it contains type and subtype attributes, together with getter and setter methods. A *Subscription* contains data concerned with the relationship of subscriber and event package: a creation time and an expiry time, the document version number, the subscription state, a reason string, the user URI, the subscription dialog identifier, and the event name. Additionally, a method to check for subscription expiry is defined. In order to be able to distinguish subscription refresh requests from new subscriptions, the equals method is redefined: A subscription is equal to another subscription if their subscriber URI and their dialog ID are equal.

The abstract class *StateChangeEvent* provides for a common abstraction of state changes. The notification server then has to define a concrete event if state changes have to be handled. For example, addition and deletion of conference resources can be signaled by a concrete class derived from StateChangeEvent. These concrete classes each define their own commands and data sets, which then have to be transferred into notification documents by the concrete EventPackage instances.

8.4 The Grappa Group Management Application

We already have described the interface to the underlying SIP stack as well as a number of concepts that are shared among group management and costream client implementation. In this section, the specific implementation details of the Grappa group management are described.

Grappa implements two kinds of SIP user agents: a conference factory UA and a focus UA. While there is only one conference factory UA, a number of focus UAs may exist. The conference

factory UA offers its service permanently, whereas the focus UAs are bound to relatively short-lived ad-hoc conferences in our case. Along with the focus UA, a notification server offers notifier support for concrete event packages which are described in the following subsection. We also present details on the management classes for collaborative groups. Finally, the GUI and the service discovery classes are briefly described.

8.4.1 Conference Focus Classes

Since Grappa should be able to serve several costream groups on demand and create them on-the-fly, we provide for two implementations of the UserAgent interface class: The *ConferenceFactoryUA* handles INVITE requests to the conference factory URI, creates a *FocusUA* and sends the SIP URI of this focus UA in the `Contact` header of a `302` redirection response. The FocusUA is registered at the SipListenerApplication as already described in section 8.3. It implements `processRequest()` and `processResponse()` and manages a map of handlers, each of which is responsible for handling all requests and responses of a particular SIP method. We derived the handler classes from the abstract class *Handler*.

The FocusUA must also handle requests for sidebar conferences. Sidebar conferences have their own URI, thus this URI is registered with the parent FocusUA at the SipListenerApplication.

The notification server of Grappa serves as a notifier for the conference, refer and streamstate event packages. For each package, concrete classes derived from StateChangeEvent are defined which model the state change commands and their data. Each event class holds a set of command names, from which the active command can be set. The *ConferenceEvent* class consists of the commands for adding and removing participants, for adding services, and for adding and removing sidebars. Up to now, we do not have a command that removes the collaborative streaming service, because there is no SIP signaling method for this. The *ConferenceEventPackage* is capable of full and partial notifications. Thus, if a state change according to the ConferenceEvent happens, a partial notification can be sent to all subscribers.

The *ReferEvent* has the three commands *pending*, *fail*, and *success*, which depend on the response that is sent by the refer target. The *ReferEventPackage* can be kept simple, because the `message/sipfrag` documents as required by the standard only contain the status line [151]. Furthermore, subscriptions do not have to be managed, because only the referrer is notified about the status. However, the subscription must be terminated, i. e. the state and reason headers must be set accordingly.

The *StreamstateEvent* has also three commands: *changeState*, *addStreamstate* and *removeStreamstate*. The latter two correspond to adding and removing sidebars in a conference. The *StreamstateEventPackage* only supports full notifications at the time being.

8.4.2 Group Management Classes

In Grappa, there are two kinds of management operations: those concerning membership management and those for managing collaborative streaming. Since membership management functionality can be used for media conferences also, we separate these two kinds of operations and implement specific services like collaborative streaming in their own classes.

The membership management of Grappa is implemented in the *MembershipHandler* class, methods of which are called from the Invite-, Refer-, and ByeHandler classes. If execution of

these operations is permitted by the membership policy, the participants or services are added or removed as required. We have implemented this intermediate MembershipHandler class to fully abstract from the SIP handling of the Focus and its Handler classes, which accounts for testability.

All conference-related entities extend the abstract *Resource* class. A *Conference* is thus a resource which has a number of child resources, i. e. participants, services, and (sidebar) conferences. The child resources are saved in a flat map, so that they can be easily accessed by their URI as keys. All (top-level) conferences are saved in the so-called *ResourceList*, which can serve as input to a graphical viewer on the set of conferences.

The collaborative streaming management of Grappa is implemented in the *CostreamManager* class, which extends the *ConferenceService* class and implements the *CostreamData* interface as shown in figure 8.8. This interface provides for storage of the state of a streaming presentation as

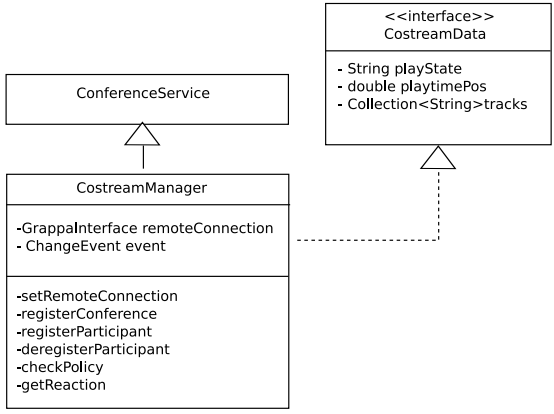


Figure 8.8: Costream Manager

delivered as a result of registration of participants at the Association Service. This state serves then as data input to the *StreamstateEventPackage* handler class. Hence, the CostreamManager is responsible for acquiring streaming state at the time a presentation is registered, and for managing this state according to the chosen group policy.

The CostreamManager is thus added to a conference like a usual service, at the time a creator or administrator registers a streaming presentation by sending a REFER message. In turn, the *registerConference()* method is called, which registers all participants to the Association Service. Later joining participants can be registered using *registerParticipant()*. Both methods return the state of the streaming presentation (which may or may not be started at the time of registration). This state is distributed to the clients using the streamstate event package. Clients subscribe to this event package either at a successful refer notification, or at the notification of the collaborative streaming service in the conference event package, using the Service URI. Since the CostreamManager is also responsible for the policy query and the appropriate reaction, it can distribute the new state in case the query was successful.

Testing a desired control action against the policy is simple, since the CostreamManager holds a reference to its *CostreamPolicy* which provides for a testing method *isAllowed()*. The *getReaction()* method must retrieve the reaction from the policy. We distinguish two reactions only: changing the state and changing the association. In the first case, the `onStateChange` method of the StreamstateEventPackage is called with an event containing a *changeState* command. It is parameterized with the information on the desired action (pause or change of play-time position). In the second case, the CostreamManager class must invoke the creation of a sidebar in the parent group. It also calls the `onStateChange` at the StreamstateEventPackage with an event containing the *addStreamstate* command and the parameters of the new sidebar association.

8.4.3 Discovery Modules

For simplicity, we use the Java API of the OpenSLP implementation to register Grappa to the SLP directory agent SLPd, which is running externally and must be started before configuration. An enhancement to a SIP proxy to register with an SLPd has been implemented in a diploma thesis at our institute [89]. We use a part of this implementation to register Grappa as a `sip:conference` service. The *ServiceAgentConfiguration* class sets SLP attributes like abstract and concrete service type or host name and port according to the configuration. The *ServiceAgentCommunicator* retrieves an *Advertiser* from the *ServiceLocationManager*. The latter two classes are implemented by the OpenSLP Java API. With the Advertiser instance, the service can be registered and deregistered after setting the configuration properties if applicable.

Grappa can also discover services itself, for example it also requires a SIP proxy for call routing. The ServiceLocationManager can provide for a *Locator*. This Locator instance is used by the *SlpUserAgent*, which finds services according to property settings.

In a static environment, discovery does not have to be run, since the configuration can be retrieved from the file system as described in section 8.3.2.

8.5 Client Implementation

The client implementation has been designed as a SIP User Agent application, which calls the RTSP streaming client as an external process. Thus, an existing media player implementation in C/C++ (such mplayer) can be used. It is also possible to use other media players using a specialized implementation of the *CostreamApplication* interface as described below.

The architecture of the client implementation has been shown in section 7.5.

8.5.1 SIP User Agent Classes

As already mentioned, the SIP functionality of the costream client is similar to the conferencing focus functionality. Our client implementation provides for a *ClientUAFactory*, which creates the required *ClientUA* implementation of the UserAgent interface.

The notification server of the costream client only has to support the dialog event package as a notifier. Our implementation for the dialog event package sends notifications on subscription requests only, since only the current state is relevant for the pull transactions. Thus, a concrete StateChangeEvent for the dialog event package is not necessary. The *DialogEventPackage* class constructs the dialog-info document using the local and remote target information.

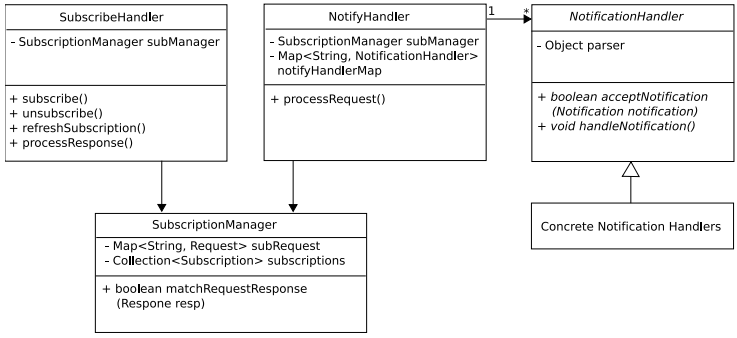


Figure 8.9: Event Notification Handling on Subscriber Side

The classes responsible for event notification on subscriber side are shown in figure 8.9.

The client must be able to manage event subscriptions, i. e. to match SUBSCRIBE responses to outgoing requests. Therefore the *SubscriptionManager* class saves outgoing requests, which the *SubscribeHandler* can match with incoming responses. In case of successful matching, the SubscribeHandler adds a subscription to the SubscriptionManager. Since subscriptions are terminated by a NOTIFY request with a subscription state of *terminated*, the *NotifyHandler* also has to save a reference to the SubscriptionManager. The intrinsic task of the NotifyHandler is to process notifications. Event package handling is usually done by concrete classes derived from the abstract *NotificationHandler* class in two stages. First, the document is parsed with a specific parser and the parsed information is put into a container class suitable for the notification document. If parsing is successful, the required data is extracted from the container class, and a CostreamEvent, which is signaled to the GUI (confer section 8.5.3 also), is generated.

An exception is the refer event notification: A `REFER` request may have been initiated by a Push or a Move action, which react differently on the refer notification, because the client must leave the group in case of a Move action. Since parsing the refer event notification is rather simple, a specific notification handler and a parser are not implemented. Instead, an *Accepted* or *Failure* event (confer also section 8.5.3) is generated, which is directly processed by the calling costream action.

The conference event package allows for partial notification. Hence, a class *LocalConferenceInfo* as an extension of the *ConferenceInfo* document container class is used to compare the saved state to the received state. While resources can be added easily, changing or removing resources requires step-by-step comparison of all entries. LocalConferenceInfo thus defines collections of changed and removed users and sidebars, and a method to commit all changes after processing the whole document.

8.5.2 User Actions

In order to abstract from the GUI where the interaction with users is done with buttons or dialogs, we have implemented user actions as mediators between GUI and User Agent classes. All user actions are derived from the abstract class *CostreamAction* as shown in figure 8.10. The actions store references to the UserAgent and to the GUI class.

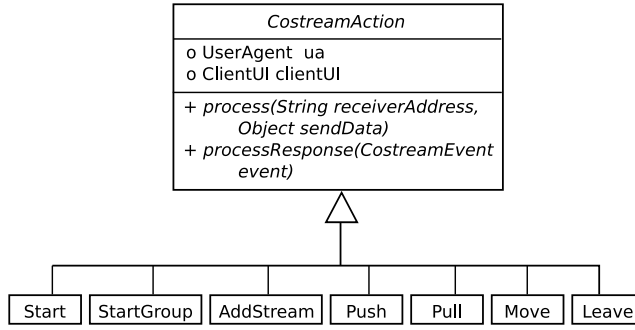


Figure 8.10: Hierarchy of User Actions

Each CostreamAction must implement the *process* method, which calls the appropriate send method of the User Agent. The receiver data and possibly additional sender data are submitted as parameters. The sender data may be processed according to guidelines of the SIP request method: For example, if a recipient list has to be delivered as content body of a request, the XML tags are added to the addresses in the *StartGroupAction*.

Each CostreamAction also implements the *processResponse* method. This method is intended to invoke sending of further requests on the User Agent or to signal the final result of the action to the initiator and, if applicable, to the RTSP client interface. The User Agent must save a reference to the concrete CostreamAction to enable the callback to this method. Thereby, *Push* and *Move* actions can be distinguished. In case of the Move action the collaborative streaming session is left. Hence, the processResponse must initiate sending a BYE request at the User Agent and a TEARDOWN at the streaming client.

Since the response data of each CostreamAction differs a lot, a common data type of *CostreamEvent* is used as a parameter to the processResponse method. This CostreamEvent can then be processed in a suitable fashion, which will be described in the following section for each type of CostreamEvent. We also use this data type for signaling events outside of CostreamActions, e. g. at a notification or at the arrival of a push request from another client.

8.5.3 Costream Events

The *CostreamEvent* abstract class is used in both the Grappa and the costream client application. Whereas it is used mostly for logging in Grappa, it has a number of applications in the costream client. Similar to the CostreamActions class, it must provide for an abstraction of UserAgent and GUI. Additionally, control actions of the RTSP client must be initiated by some of the CostreamEvents.

Figure 8.11 shows the hierarchy of receiver events. All events have three attributes: First a type, which denotes the method corresponding to the event, second a string parameter which denotes the collaborative group the event is related to, and finally certain data the event can operate on. Each event implements the abstract *handle()* method. Depending on the configured user interface, the handle method may signal the event to a user or log the event.

For some events, the handle method may require user input or initiate actions on the RTSP client.

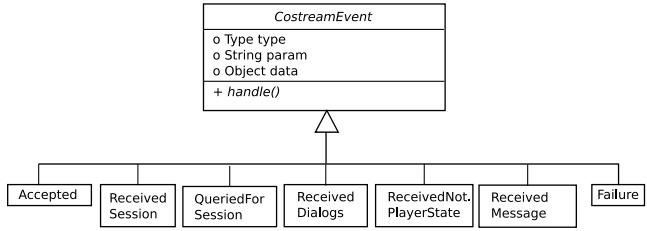


Figure 8.11: Hierarchy of Receiver Events

- The *ReceivedSession* event signals incoming push requests. The user is asked for acceptance. After sending the response back to the caller, the event initiates the start-up of the RTSP streaming client, if appropriate.
- Acceptance is also required by the *QueriedForSession* event, which signals a pull transaction incoming from another user. The client or the human user must decide whether the dialog identification of the collaborative group session can be given to the caller.
- The *ReceivedDialogs* event signals incoming group URIs as a result of the Pull transaction, which have been extracted from the dialog-info document by the DialogNotificationHandler. The user is asked to select one group URI, for which a join with a usual SIP INVITE will be tried.
- The *ReceivedNotificationPlayerState* event operates on the data set of the streamstate event package. It is used to signal streaming state changes of the association to the RTSP client, which cannot be done by means of standard RTSP. According to the contents, actions on the RTSP client interface are called, e. g. buttons are enabled or disabled, or the slider position is adapted.
- The *Accepted* event signals the successful submission of a SIP request. We also use it to signal the positive completion of a costream action, which is useful in case of actions that use the `REFER` method: The initiator of the request gets the acknowledgement that the REFER has been received and is processed, but this is no final guarantee that the refer target accepts the invitation. This final acceptance by the REFER target is normally delivered in a refer event notification and can be handled in the corresponding action as shown above. In the exceptional case that the refer event notification is suppressed, the information can be taken out of the conference event package.
- The *Failure* event signals failures of operations to the user, possibly with information on the failure as given by the reason text in the SIP response. It can be used for all kinds of failures.
- The *ReceivedMessage* event signals incoming messages. In our case, this event is used only in the special case that a conference is ended on some other user's demand. It could also be used for further SIP requests (INFO, MESSAGE, OPTIONS, ...), which are less relevant for collaborative streaming.

Although the *ReceivedNotificationMembership* has been classified among CostreamEvents in table 7.1 in section 7.5.1, we decided not to implement the conference event notification as a CostreamEvent. Instead, we have implemented a ConferenceNotificationHandler, which delivers

the necessary data to the Group directly. This has been done for the following reasons: First, conference events occur quite often, so signaling every event would disturb the user. Second, the conference event must be compared to the information already available in the Group, therefore handling a specific event would introduce another level of indirection.

CostreamEvent objects are initialized by the User Agent. The User Agent can then either call the *handle* method or pass the event object to the appropriate costream action (which has been saved as a reference before). CostreamEvents thus are used as a data container on one hand, and as an implementation of the *Command* pattern [44] on the other hand.

8.5.4 RTSP client interface

We have implemented an additional GUI to the available mplayer to be able to control the client from our client application. This is possible because mplayer can be controlled by keystrokes, which can easily be synthesized from GUI commands.

Since a costream client can stream several presentations, a *SessionManager* is used to map collaborative groups to sessions. For simplicity, we use a 1:1 mapping of sessions to streaming clients. Therefore, the SessionManager also maintains a mapping from streaming clients to sessions, which enables callbacks from the streaming client. For example, a session can be finalized in case the streaming client is terminated.

The *CostreamApplication* interface provides for access to its particular RTSP client from the SessionManager and/or human users. Users access the client by the load, start, suspend, resume or stop operations. The play-out position can be changed by the setPosition operation. The hierarchy of the streaming client implementation is shown in figure 8.12.

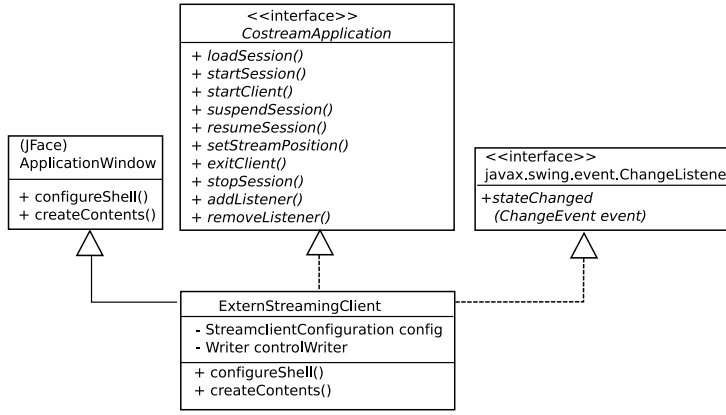


Figure 8.12: Hierarchy of the Streaming Client Interface

The *ExternStreamingClient* class provides for an implementation of the CostreamApplication interface. For simplicity, it starts the mplayer client as an external process. In order to be able to control this external process, an implementation of a *CommandRunner*, originally invented by the authors of the `java.util.concurrent` package, is used. This CommandRunner synchronizes

input and output data streams using a *CyclicBarrier*. The performance of this class has been found sufficient in the sense that a difference to running the stand-alone mplayer client could not be perceived. Hence, this class has not been modified. The ExternStreamingClient maps the above-mentioned interface control operations into command strings, which are written onto a piped output stream.

The RTSP client interface also provides for the setting of configuration options, for example the RTSP proxy address and the participant's SIP URI. Default values for these properties are provided by the system so as to minimize the configuration effort for the user. The configuration is stored in a file, which is read by the mplayer client before presentation start-up. The configuration cannot be changed on-the-fly, because this would require additional changes in the mplayer code.

8.5.5 GUI classes

In the client, a graphical user interface is important, however, it should be possible to run the application without a GUI on demand. The latter can be used for devices which are controlled by a third-party client. We have only implemented a simplistic GUI which allows to call the necessary user transactions, shows an overview of all sessions and collaborative groups, and a message log window for debugging purposes. Chat and conferencing functionality has not been implemented in our test implementation but can be added by implementing the necessary *MediaConfService* classes on client side also.

We have implemented the GUI classes using the SWT and JFace packages [53] maintained by the Eclipse Foundation, because SWT promises an efficient implementation with a neatly integrated user interface for many platforms. JFace is built on top of SWT and follows the Model-View-Controller pattern [128]. Since start of development, further development to the java-inherent Swing classes has been done, thus Swing seems to be another possible candidate for efficient GUIs. Since we have strived to separate actual GUI functionality from processing as far as possible (by the help of CostreamAction and CostreamEvent classes), the toolkits should be exchangeable.

For the simulation of devices, we implemented a Console-style UI which can be controlled by keyboard interactions.

In figure 8.13, a class diagram of the client UI hierarchy is shown.

We have already mentioned the *ResourceList* interface as a common data model for all resources in either the Grappa or the client application. On the GUI side, this ResourceList serves as input to a JFace *TreeViewer*. Thus, the *ResourceListComp* provides for the graphical set-up of all components forming the tree, and implements listeners to mouse events and change events. Directions on how to convert the data input of the ResourceList into tree components have to be provided by the *ResourceListContentProvider* class, which implements the *IContentProvider* interface with its `getElements` and `getChildren` methods. Each element has a textual representation provided by the *ResourceListLabelProvider*.

We have already mentioned actions and events as mediator abstractions modeling the active and passive parts of a client, respectively. Actions are mirrored at the GUI as classes extending the JFace *Action* class. Each action has a certain textual and an image representation at the GUI, and overrides the `run()` and `runWithEvent()` methods of the superclass. All actions are put into an *ActionRepository*, a static component which allows to add those actions to other GUI components as required.

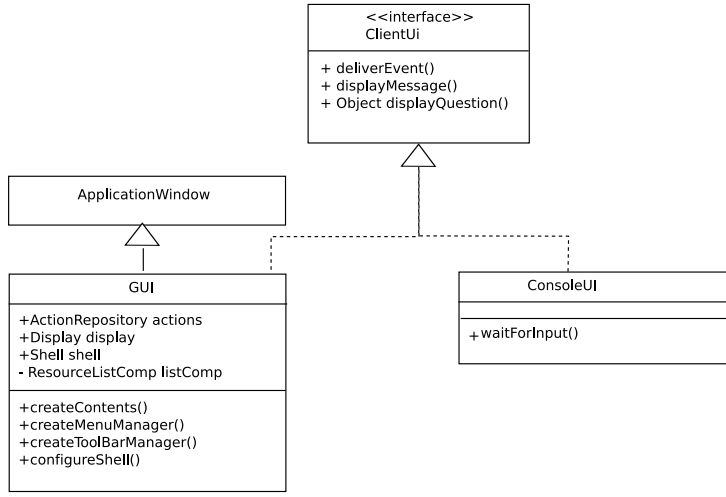


Figure 8.13: Client UI classes

Events are signaled to the GUI only in case their `signalGUI` attribute is true, since some transactions require user interaction whereas others do not. If an event requires interaction, the *displayQuestion* method queries the user for acceptance, otherwise, the *displayMessage* method of the GUI prints an informational message into a separate window. In case of a console-style GUI, events are logged to the console or to a file. Interaction is simulated using automated policies, which are quite simple in our case: Invitations are always accepted. Sophisticated policies, which may be similar to the policies used at the group management applications, can enhance the client application.

A number of helper dialogs complete the GUI implementation. For example, preferences or policies have to be edited, or individual components have to be chosen from a list.

8.6 Summary

In this chapter, we have presented a prototype implementation of our collaborative streaming architecture. The architecture basically consists of the group management application Grappa, which is implemented as a SIP conferencing focus and provides additional access control functionality for the collaborative streaming session. Grappa also registers groups at the association service implementation Cassis, which is implemented as an RTSP proxy, relaying session control requests of clients upstream to further proxies or the streaming server. Cassis also implements session sharing, i. e. the management of associations, and synchronization. The synchronization module implements the independent and the reflected synchronization mode.

The collaborative streaming client application Cream combines a SIP user agent for session transfer requests and a front-end to an external media player (mplayer in our prototype), which is used as an RTSP client for session control. We also designed and implemented a number of client events. These provide for an abstraction of results from session transfer methods which are signaled to the session control front-end or to the user interface.

A discussion and a quantitative evaluation will be presented in the next chapter.

9. Evaluation

In this chapter, the concept of a collaborative streaming architecture is evaluated in a qualitative and a quantitative sense. First, the conformance to the requirements presented in section 2.5.1 is discussed. Afterwards, the latency introduced by the association service *Cassis* and the group management application *Grappa* is examined for all initiation and update transactions. Moreover, the synchronization styles of *Cassis* are examined in more detail how far they can synchronize a late-joining client to a running presentation. Finally, the message overhead introduced by the costream architecture is reviewed in comparison to standard streaming and conferencing sessions.

9.1 Qualitative Discussion

The following *Session Transfer* primitives have been implemented by our concept: Start, StartGroup, Push, Move, Pull, and Leave. The Push and Pull primitives are usable in a synchronized form, which means that the joining client is added to the particular association of the already active transaction partner. Additionally, an unsynchronized form of Push and Pull allows clients to be referred to the streaming presentation without actively joining the collaborative group. The Start and StartGroup transactions can be executed before or after the start of the streaming presentation by the initiator. This is due to the association state machine, which allows registration of an association after setup. The Leave transaction can be used to leave both group and streaming session, or to leave the group only while keeping the streaming session. In the first case, the group management deregisters the member from the association service completely and streaming is stopped. In the latter case, the relation of participant and group URI is released by means of the deregistration, but the member gets a new association which keeps the streaming session.

We have used standard SIP and its extensions for the implementation of session transfer. Though inconsistencies with RFC standards have been avoided, some concepts have been used differently than envisioned by standards documents:

- According to RFC 3515 [151], the URI in the Refer-To header can be a non-SIP URI. In our architecture, an `rtsp://` URI is used to register a streaming presentation. It is proposed in our concept that the callee (i. e. the group manager) sends a final success notification after registration of the streaming session to the association service. The procedure of registration belongs to the core functionality of the group management. The focus must be able to process the registration request. If the association service is not connectable, no collaborative group can be set up and a failure notification will be sent. The SIP dialogs with conference participants will however persist until the participants close them or the conference is removed.
- Service URIs in the conference event package (RFC 4575 [137]) describe additional conference-related services, like web pages for information about the conference. In the

costream architecture, the streaming presentation is also added as a service-URI in the conference event package, with the newly defined purpose value of `collaborative-streaming`. Clients that are not able to process this cannot automatically start a streaming session. Instead, a human user would have to supply the streaming session URI to the player application manually.

- A policy document has been added to message bodies of the INVITE transaction. This means that the focus has to accept and send multipart/mixed MIME bodies containing both the session description and the policy, at least in case it is desired that clients can change policies. The session description is processed according to the usual rules of the offer/answer model (RFC 3264 [133]). The policy sent by the client is always accepted if the document conforms to the rules of a policy document as described in appendix B. The handling of the policy body is optional, i. e. any component that cannot process the body ignores it. If a focus does not support processing of policy bodies, the group management still can use static policy configurations.

Furthermore, a streamstate event package has been added, with the document type of `application/x-streamstate-info+xml`, to avoid problems with existing parser implementations. A client that cannot read this package will, however, not be able to receive notifications about control requests.

Session Control has been designed for the control of both the play-time position and of the track selection. In the policy document, `Pause`, `ChangePlaytime`, and `ChangeTracks` are the defined permissions. Roles, which own one to many permissions, have been added to map those permissions to members in an efficient but flexible way. Latency measurements of play-time control operations will be presented in section 9.2.2. The other clients are notified about successful control operations by a streamstate event and – if a new association has been opened – by the conference event package.

Session Sharing is implemented by the association service, which synchronizes joining members of an association. We have implemented reflected and independent synchronization. The reflected synchronization mode uses an external reflector, because this entity has already been available. We did not implement feedback optimization because it requires synchronized clocks and more complex operations in the client. The quality of the synchronization will be discussed in section 9.2.3.

SLP service discovery has been added to the *configuration* of the applications. The lookup of the collaborative streaming session at a pull transaction is done by subscribing to the SIP dialog event package. For address lookups, a register where entries can be added manually is used. To enable privacy, the subscription to the SIP presence event package [131] has to be integrated with the use of registration information from a SIP registrar. Additional mechanisms to provide for anonymity used by SIP do not make much sense for collaborative streaming, because it is assumed that collaborative group members have a tight relationship.

A *discussion channel* for group communication can be added by integrating an interactive audio/video or chat application with the client. The offer and answer session description bodies of the INVITE transaction messages will additionally carry media descriptions in this case. Those media descriptions have to convey media data transport and format information.

Transport efficiency can be achieved by using the reflected synchronization mode for environments where users are expected to watch a large part of the presentation together, i. e. few individual state

changes occur. We did not conduct experiments with caching or transcoding, but these features have been added to the beaver reflector implementation in a parallel project and can be used in the reflected synchronization mode, possibly with client modifications to signal transcoding requirements [13], [96].

9.2 Quantitative Evaluation

For the quantitative measurements a certain deployment of the clients as well as of the costream components is required. A large number of possible settings are conceivable, from which we have chosen four settings, which provide for a compromise between emulation of the application scenarios presented in chapter 2 and testability.

Basically, the costream components and the clients have been distributed among two networks in an experimental testbed. The *access* network is a 100 Mbit/s switched Ethernet LAN behind a 2 Mbit/s downstream and 512 kbit/s upstream ADSL connection. A 1 Gbit/s Ethernet LAN with switches capable of handling Gbit traffic has been used in order to simulate the placement of costream components in a *service provider* network. The Dual-Xeon Linux server *radiator* has been chosen to run costream components where necessary. Another Dual-Xeon server called *streaming* runs a Darwin Streaming Server version 5.0.

Average round-trip delays for a ping from the access network clients to *streaming* and *radiator* have been about 25.8 ms each, with a mean deviation of 0.47 ms (short-scale measurement). In the NIST SIP stack, UDP has been selected as a transport protocol, because problems exist with TCP regarding SIP dialog management. Though slower machines often trigger retransmissions, experiments showed no big latency difference of TCP and UDP.

The evaluation scenario settings that have been set up emulate the scenarios presented in chapter 2. The *home* setting shown in figure 9.1 runs the clients in the access network and the intermediate components (*Grappa* and *Cassis*) in the provider network.

Another possibility to set up a home scenario is to deploy the costream components within the access network. The latency values are lower than in the setting depicted above. However, the management gets more complicated and security risks may arise if clients connect from outside.

The learning scenario is reflected by the *university* evaluation scenario shown in figure 9.2, where all components reside in the service provider LAN to simulate a high-bandwidth and low-delay environment. In realistic environments, the streaming server may be on the global Internet. However, it is assumed that a learning institution has a broadband connection to the Internet. Thus, only a certain extra delay has to be added for each operation concerning the streaming server.

In figure 9.3, the *distributed clients* scenario is shown, which simulates clients within different networks. In our case, one client resides in the access network, the other one is put into the service provider network. Such a setting can appear in learning environments in case that remote or mobile clients connect to the system.

The spontaneous meeting scenario could not be truly emulated by this distributed client scenario, because in wireless environments, much more errors appear than in fixed network environments. Hence, the spontaneous meeting scenario is similar to the distributed clients evaluation scenario, but operations that involve the mobile client would suffer from higher latency and more transmission faults.

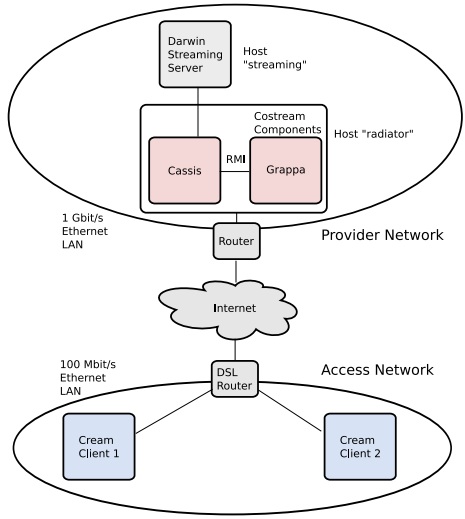


Figure 9.1: Home Evaluation Scenario Settings

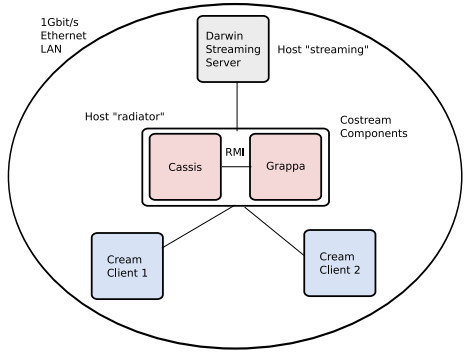


Figure 9.2: University Evaluation Scenario Settings

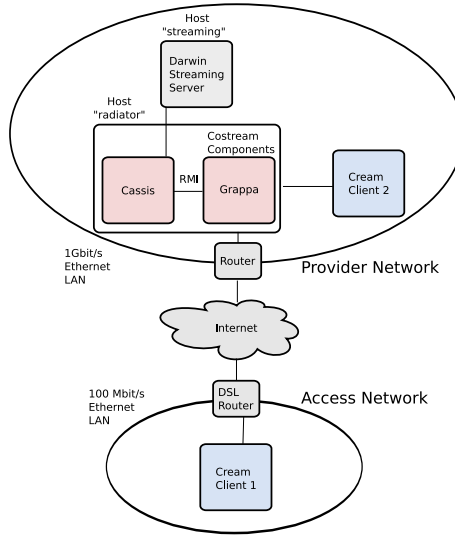


Figure 9.3: Clients Distributed on Separate Networks

In some cases, particularly for push/pull and update transactions, we have also conducted measurements where the costream components have been distributed onto the access and service provider networks, as shown in figure 9.4, so as to simulate the distribution of costream components.

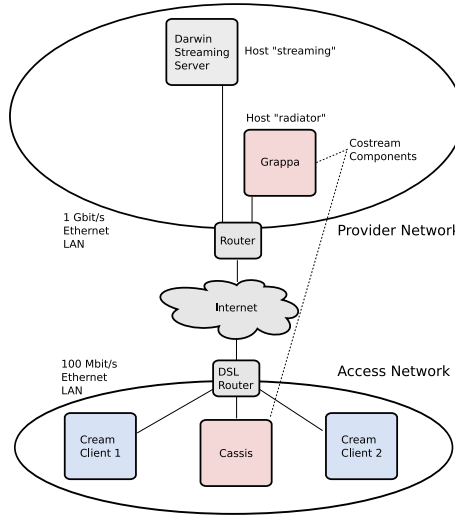


Figure 9.4: Costream Components Distributed on Separate Networks

For the quantitative measurements, a specific test control client has been designed. This client is based on the Cream implementation, but does not have a graphical user interface. Instead, the transactions that have to be measured and their parameters are chosen by command-line options. The interface to mplayer is also simplistic: The so-called *ConsoleStreamingClient* provides access to mplayer's control functions without GUI. Furthermore, the audio and video drivers of mplayer are set to null to suppress output and simplify control interaction. The control client allows to run a certain number of iterations for the transaction that has to be measured.

9.2.1 Latency Measurements for Initiation

The goal of these latency measurements is to show that for each initiation transaction, the latency introduced by the costream architecture is low enough not to disturb users.

The latency for the different initiation transactions has been measured by collecting a packet trace of SIP and RTSP requests and comparing the sending time of the starting request (INVITE, REFER, or BYE) with the reception time of the final response or notification. For all initiation transaction measurements, the independent synchronization mode of the association service has been used where applicable. In case the reflected synchronization mode is applied lower latency values can be achieved in realistic environments.

9.2.1.1 Start Transaction

The time required for the start of a collaborative group is measured by calculating the time difference from sending the the INVITE to the conference factory to the reception of the conference package notification, which contains the presentation URI. In figure 9.5, this is shown as the *Group Setup* time.

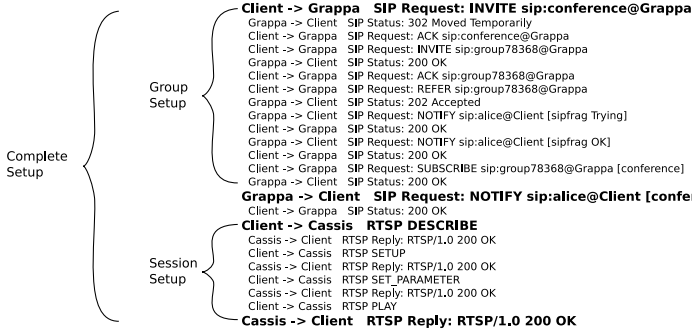


Figure 9.5: Measurement Points for Start

Additionally, the time required for streaming session setup is measured (*Session Setup* in the figure) as well as the time for the complete transaction. The difference between the complete setup time and the sum of group and session setup times consists mainly of the time for starting the media player application. The necessary subscription of the streamstate event package is done after the successful setup of the streaming session and is therefore not considered in the measurements.

During the experiments, we have noticed that the first iteration always has taken a larger time than the rest of the iterations. For example, the complete setup for the university scenario has been up to one second if all components had been restarted before, whereas the latency for the rest of the iterations has been lower than 210 milliseconds for 90 percent of the iterations, as also shown below. Several factors contribute to this behavior: First, the NIST SIP stack requires initialization of its data structures. An example application provided along with the implementation [118] has performed similarly during our measurement experiments. Moreover, the costream client prototype implementation has not been designed in an optimized fashion regarding latency. Particularly, the initialization of the external mplayer process takes more time at the first start of this application. In case that the costream intermediate components have been restarted, the latency for streaming session setup itself is also higher, because the association service has to resolve the name of the streaming server.

Another factor which adds to the larger delay of the first iteration is the necessary registration of the group at the association service. However, the difference among the first and the forthcoming registrations is quite small. All in all, the communication among the costream components using RMI takes only a few milliseconds (measured inside the application) if the costream components are on the provider network, and less than 150 ms if they are distributed among access and service provider network.

Since the costream intermediate components are expected to run without restarting them in a realistic environment, and the other deficiencies of the prototype can be eliminated in an optimized implementation, the latency of the first iteration of any run will not be considered any further in the following measurements. This means that in each run of an experiment only the last 10 iterations are collected for the statistics.

The results are shown for the university and the home scenario. For start transactions, the scenario with distributed clients has no meaning, because only the initiator client is involved. Hence, either the home or the university scenario applies, dependent on where the initiator client resides. The distributed costream scenario is omitted here, because it shows different behavior dependent on the network in which each component is placed.

A histogram of all latency values for 20 of such experiment runs illustrates the differences of the individual scenarios, as shown in figure 9.6. The values have been classified according to their latency using steps of 100 ms up to a value of 1.0 s.

In the university scenario, the group and session setup times are always below 0.1 s. Hence, the complete setup lies mostly between 0.1 s and 0.3 s, with few exceptions up to 0.5 s. In these cases, the start of the external player process at the client machine took longer time. In the home scenario, most group and session setup times are lower than 0.4 s, but again there are few exceptional values up to 1.0 s. Besides the mentioned problems with starting the external player process, varying networking delays due to the DSL connection have to be considered in this scenario. However, 90 percent of the latency values of the complete setup are lower than 0.7 s, as also shown in table 9.1 together with other statistical values for the university and home scenarios. All in all, these values show that the start of a collaborative streaming group is done in an acceptable time period even for the home scenario.

These latency values are valid for start transactions which have the initiator as the only member. If a StartGroup operation is chosen, the latency of the push transaction as described in the next section has to be added.

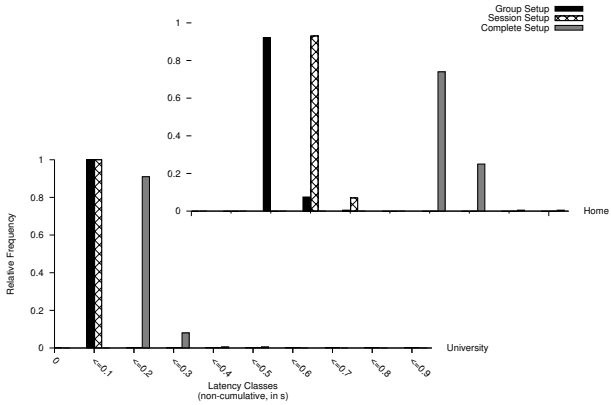


Figure 9.6: Histogram of Start Transaction Latency Values

Setting	Function	min (s)	max (s)	avg (s)	mdev (s)	90th perc. (s)
University	Start of Group	0.03	0.09	0.04	0.01	0.05
	Streaming Session Setup	0.02	0.07	0.02	0.01	0.03
	Start (Complete)	0.11	0.46	0.15	0.04	0.20
Home	Start of Group	0.17	0.39	0.19	0.02	0.20
	Streaming Session Setup	0.25	0.39	0.29	0.01	0.30
	Start (Complete)	0.54	0.90	0.59	0.04	0.66

Table 9.1: Latency Measurements for Start Transaction

9.2.1.2 Push Transaction

In the experiments for measuring the push transaction latency, it is interesting to examine the views of both the caller and the callee. In figure 9.7, the measurement points for the push are shown.

Figure 9.7: Measurement Points for Push

The setup of the callee is similar to the setup of any collaborative streaming session. For the caller, the time between sending the REFER request and receiving the success notification (SIP NOTIFY with sipfrag 200 OK) is essential. The home and university scenarios are the same as described in the last section, except that a second client has been added. In the scenario with distributed costream components, we have put the clients and the association service into the access network, whereas in the distributed clients' scenario only the callee resides in the access network.

In figure 9.8, a histogram of the values the setup takes for the callee is shown. Additionally, the latency as perceived by the caller is depicted for group, session, and complete setup.

The distributed clients scenario shows similar characteristics like the home networks setting, because the callee has been put into the access network. In the home network, the session setup times of the callee are slightly higher because of the fact that the caller is already streaming media data in the home network, which utilizes the networking connection to the provider and causes waiting times. The distributed costream scenario has a larger group setup latency than the other settings, because Grappa and Cassis are on different networks, and the registration has to be sent to Cassis in a synchronous fashion so as to confirm that the collaborative streaming service is available. However the session setup is slightly faster than done in the home and distributed client scenarios. This results from the fact that Cassis is put into the access network, which provides for a faster route to the streaming server. If requests have to be sent via Cassis running on the host radiator, these packets take a more indirect way to the streaming server. This experiment also shows that in a productive environment, the association service component should be placed in a deliberate fashion.

In table 9.2, the most important statistics are collected for the different scenarios. The time until the caller is notified of the success of its REFER message is denoted in the *Caller Perception* rows. For the callee, the results for group and session setup as well as the results for the complete setup have been denoted.

The push transaction requires human interaction in the form that the callee must accept the invitation. In this measurement, only the latency introduced by technical components is considered.

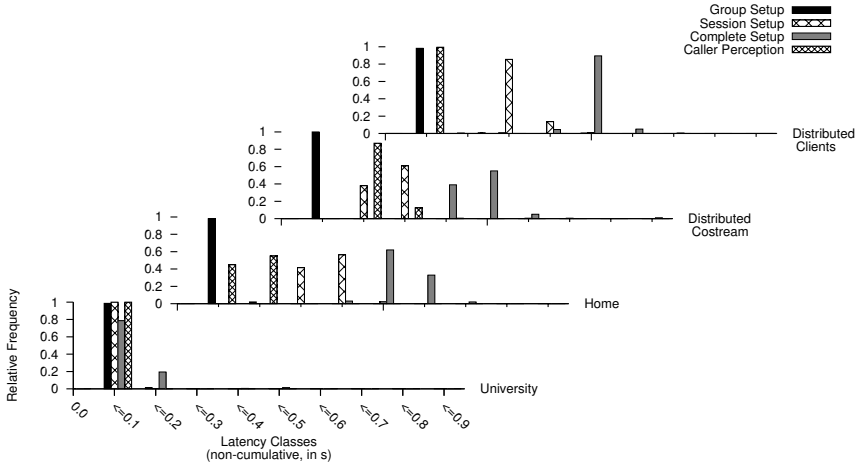


Figure 9.8: Push Transaction

Setting	Function	min (s)	max (s)	avg (s)	mdev (s)	90th perc. (s)
University	Caller Perception	0.01	0.01	0.01	< 0.01	0.01
	Callee Group Setup	0.01	0.06	0.01	< 0.01	0.01
	Callee Session Setup	0.01	0.04	0.02	< 0.01	0.02
	Callee Complete Setup	0.08	0.39	0.10	0.03	0.14
Home	Caller Perception	0.09	0.16	0.11	0.02	0.14
	Callee Group Setup	0.06	0.11	0.08	0.01	0.10
	Callee Session Setup	0.21	0.44	0.31	0.04	0.36
	Callee Complete Setup	0.38	0.66	0.48	0.05	0.55
Distributed Costream	Caller Perception	0.15	0.32	0.18	0.02	0.21
	Callee Group Setup	0.12	0.29	0.14	0.02	0.16
	Callee Session Setup	0.18	0.62	0.21	0.04	0.24
	Callee Complete Setup	0.35	0.85	0.42	0.06	0.47
Distributed Clients	Caller Perception	0.04	0.13	0.05	0.01	0.05
	Callee Group Setup	0.05	0.13	0.06	0.01	0.06
	Callee Session Setup	0.24	0.48	0.28	0.03	0.31
	Callee Complete Setup	0.38	0.62	0.43	0.03	0.48

Table 9.2: Latency Measurements for Push Transaction (REFER sent to Focus)

If human users are involved, latencies rise because of the reaction times of humans, which are difficult to forecast. Since the technical latency is lower than one second in nearly all cases, users will not be irritated by long waiting times, which proves the user-friendliness of the costream concept.

As presented in section 7.2.2, the REFER method of the push transaction could also be sent to the callee. We have also conducted measurements for this, but found the results very similar to those presented above. With respect to latency it does not matter which version to prefer. However, implementors may explicitly require to send REFER via the focus.

9.2.1.3 Leave Transaction

We did not evaluate the move transaction, since the processing is similar to the copy examined in the last subsection. The latency perceived at the callee side is equivalent; only the values for caller perception must be increased by the amount of the leave transaction latency, which is shown in the following.

The latency for the leave transaction is measured as the difference of sending the BYE request and reception of the OK or the NOTIFY request sent after unsubscribing from the conference event package. The distributed clients scenario has not been examined for the leave transaction, because the leave affects only one client, whereas the other members receive an asynchronous notification.

In figure 9.9, a histogram of the latency values for a leave transaction is shown.

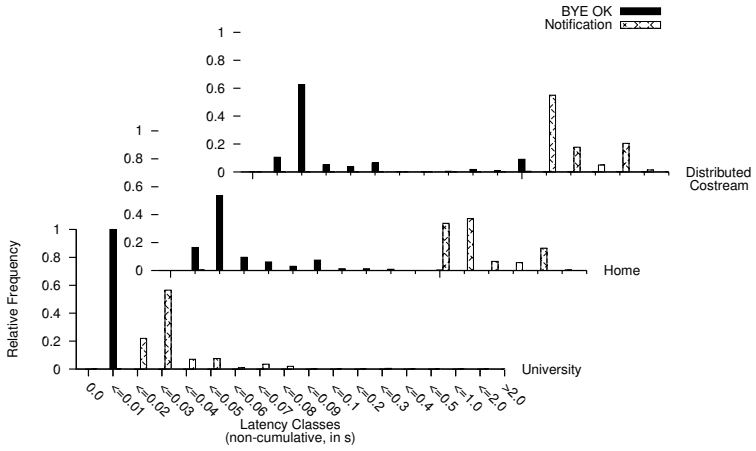


Figure 9.9: Leave Transaction

The BYE OK bars shown in dark gray denotes the time from sending the BYE request until reception of the 200 OK response. For the human user, this feedback is enough to confirm leaving of the group. However, the SIP stack will exchange a final SUBSCRIBE/""NOTIFY for

unsubscription of the conference state. The time until this exchange is finished is denoted in the Notification bars. In the home and distributed costream scenarios, the time until the final notification is received can be long. Sometimes, the tear-down of the streaming session is sent in-between the SIP message flow.

Setting	Function	min (s)	max (s)	avg (s)	mdev (s)	90th perc. (s)
University	Notification	0.01	0.14	0.01	0.01	0.01
	BYE OK	< 0.01	0.14	< 0.01	0.01	< 0.01
Home	Notification	0.05	0.79	0.12	0.10	0.17
	BYE OK	0.03	0.69	0.07	0.07	0.10
Distributed Costream	Notification	0.11	1.15	0.19	0.11	0.24
	BYE OK	0.03	0.93	0.08	0.09	0.12

Table 9.3: Latency Measurements for Leave Transaction

The statistical values for leaving a group are summarized in table 9.3. Since the notification about the client leaving a group is also sent to all group members, the overall latency is quite high. This could be optimized by sending the unsubscribe request before leaving the group. However, that behavior extends the time from starting the leave transaction until receiving a BYE OK confirmation.

9.2.1.4 Pull Transaction

The pull transaction, which "copies" a session from another user, is interesting to measure because of the necessary subscription to the dialog state of a collaborative group member. In figure 9.10, the message flow is summarized.

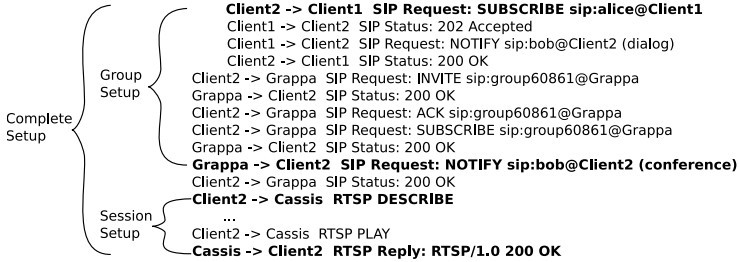


Figure 9.10: Measurement Points for Pull

The group setup is measured from the point of sending the SUBSCRIBE request to the group member until reception of the notification of the conference state. The session setup is identical to the procedure in the start and push transactions.

The latency values show the characteristics as depicted in the histogram of figure 9.11.

Due to the additional subscription/notification exchange between clients, the latencies are slightly higher than in the push case. Particularly the distributed clients scenario has a higher group

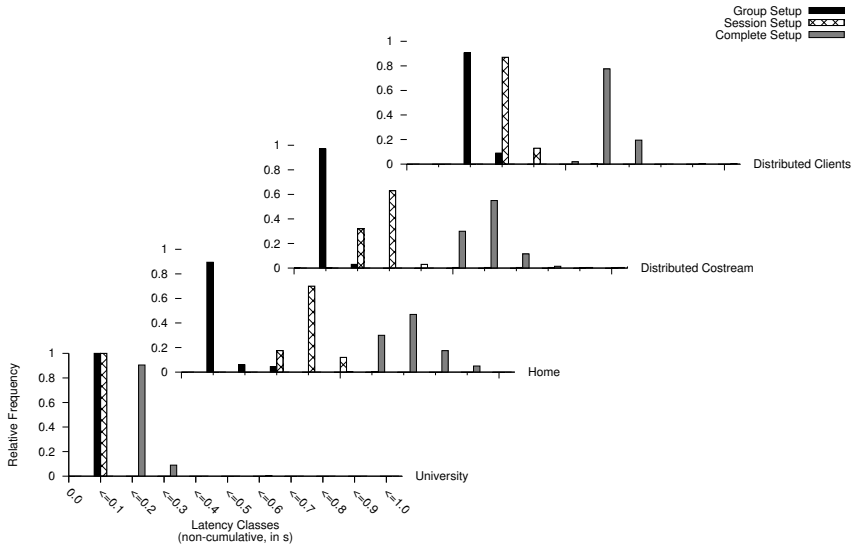


Figure 9.11: Pull Transaction

setup time, because the subscription to the client's state has to travel quite a long way. Similarly, the distributed costream scenario has quite a high setup latency because the registration at the association service needs more time. Since the association service is placed in the access network, the complete setup takes less time because of the faster streaming session setup.

Setting	Function	min (s)	max (s)	avg (s)	mdev (s)	90th perc. (s)
University	Group Setup	0.02	0.09	0.04	0.02	0.06
	Session Setup	0.01	0.06	0.02	<0.01	0.02
	Complete Setup	0.10	0.53	0.15	0.04	0.19
Home	Group Setup	0.06	0.22	0.08	0.03	0.10
	Session Setup	0.22	0.53	0.34	0.05	0.41
	Complete Setup	0.46	0.90	0.65	0.08	0.77
Distributed Costream	Group Setup	0.05	0.14	0.07	0.01	0.08
	Session Setup	0.18	0.74	0.23	0.06	0.26
	Complete Setup	0.44	1.23	0.55	0.09	0.64
Distributed Clients	Group Setup	0.17	0.52	0.19	0.03	0.20
	Session Setup	0.22	0.38	0.27	0.02	0.30
	Complete Setup	0.49	0.92	0.57	0.05	0.64

Table 9.4: Latency Measurements for Pull Transaction

In table 9.4, the statistics for the pull are collected. We assume that the client which is already a group member automatically responds to the SUBSCRIBE of its dialog state with the notification about its collaborative group dialogs.

The push and pull transactions are difficult to compare, because each of them has its own use cases. The pull transaction allows clients to join the session actively. This is advantageous in the sense that it affects the client that is already in the session to a lesser extent. The policy decision who may join one's sessions can be handled automatically. However, some scenarios may allow joining of clients only after an explicit invitation as done by a push. Moreover, the pull transaction requires the provision of the remote dialog target URI by the conference participant inside the dialog event notification, which is not mandatory.

For all transactions, using SIP provides for another advantage: Since notifications and preliminary responses about the status of a call are submitted to the caller, a user has the experience of a truly interactive application, even if the callee takes longer time for a reaction.

9.2.2 Policy Query Measurements

We have implemented the policy at the group management application, which means that the association service must query the policy for each control request. Another option would be that the group management registers the policy at the association service. The association service could decide about control requests without asking the group management, which enables faster response times. However, the group management must still be informed about possible state changes. Hence, the amount of data which is exchanged among association service and group management is larger.

In the following subsections, measurements concerning latency are presented and compared. First, a general distinction by the policy decision of the group management is made, because the different reaction policies cause different protocol messages to be sent among the components, as already shown in section 7.3. Afterwards, the influence of the synchronization mode on the latency for the state change and for the partition of the group are presented.

9.2.2.1 Influence of Policy Decision

Considering the latency values of update transactions, the update operations themselves (like pausing or changing the position) are less relevant. Instead, the reaction of the costream intermediate system determines the values of the latency.

In case that only one member is in an association, a policy query is unnecessary. The association service forwards the request to the streaming server. Otherwise, the association service has to retrieve the policy decision from the group management. According to the policy defined by the group, three different cases of reactions can be distinguished: the operation is either forbidden, or the state of the whole association is changed, or a new association is opened. In the case of a state change, the latency is measured from the time of sending the operation request (PAUSE or PLAY) until the last member of the association has received the OK response. This is done to show an upper bound for the time the state change takes to be performed for all association members. In all other cases, the other association members are not involved directly into the operation. Therefore, we have measured the time until the response on the operation method has been passed to the requester. We have used the independent synchronization of the association service for this experiment.

We did not make experiments for the distributed clients scenario, because the state change reaction is the only setting that is different from the university or the home scenario (dependent on which client issues the control request).

In the following, we show the latency values for the different policy decisions, i. e. dependent on the reaction to the control request.

In the university scenario of figure 9.12, the latencies for all operations except the state change are similar. In case of a state change, the other clients must receive the notification before they

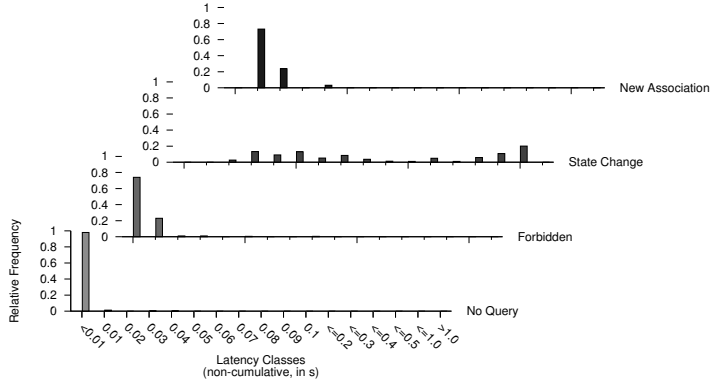


Figure 9.12: Update Latency Values in University Scenario

send their own control operation to the association service (cf. the following subsection also). This behavior leads to the higher latency, because the client application has to react properly on the SIP notification and translate it into a control operation of the RTSP client.

In the home scenario shown in figure 9.13, all operations take longer than in the university scenario because of the larger networking delay to the association service. The forbidden query only requires one policy query but no interaction with the streaming service itself. Like in the university scenario, the state change takes the longest time because of the required notification of the other member. The control request producing a new association takes shorter time than the state change, because the other member does not have to react on the operation.

Finally, regarding figure 9.14 it is visible that in the scenario where costream components are distributed, the operation without query is faster because of the smart placement of the association service. However, the other operations require longer time because the policy query experiences larger delay.

The results also have been summarized in table 9.5. Comparing the average values, the observation can be made that the amount of time required for a control request without query is not much smaller than for querying the reaction, at least if the costream components reside in a common network. Hence, we do not transfer the policy to the association service or cache it there. Note that in our case the streaming server is quite close to the costream components. Hence, in a

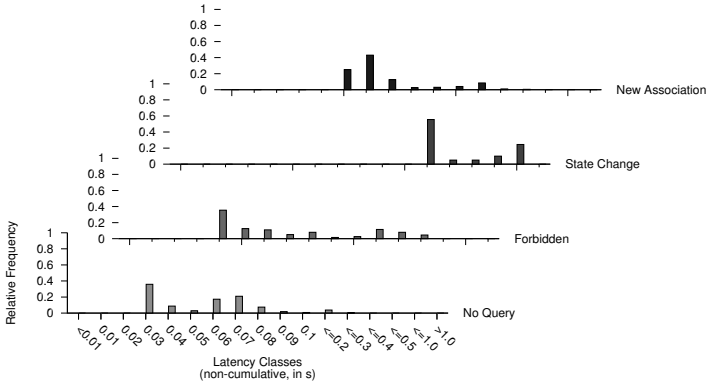


Figure 9.13: Update Latency Values in Home Scenario

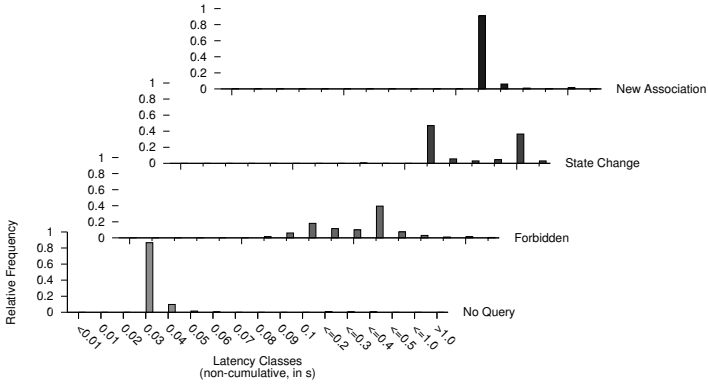


Figure 9.14: Update Latency Values in Distributed costream Scenario

Setting	Function	min (s)	max (s)	avg (s)	mdev (s)	90th perc. (s)
University	No Query	< 0.01	0.04	< 0.01	< 0.01	< 0.01
	Forbidden Operation	< 0.01	0.08	< 0.01	0.01	0.01
	Reaction: State Change	0.02	0.84	0.23	0.24	0.62
	Reaction: New Association	0.01	0.04	0.01	0.01	0.02
Home	No Query	0.03	0.25	0.06	0.03	0.08
	Reaction: Forbidden	0.04	0.37	0.09	0.08	0.24
	Reaction: State Change	0.11	0.94	0.31	0.21	0.66
	Reaction: New Association	0.05	0.36	0.08	0.04	0.12
Distributed Costream	No Query	0.03	0.37	0.04	0.03	0.04
	Reaction: Forbidden	0.06	1.19	0.14	0.12	0.24
	Reaction: State Change	0.08	1.66	0.41	0.30	0.82
	Reaction: New Association	0.11	0.93	0.18	0.09	0.20

Table 9.5: Latency Measurements for Update Transaction

realistic environment the latencies for requests that have to be sent to the streaming server may be higher, dependent on the location of the streaming server.

All in all, control requests are quite fast. Compared with the time the media player application takes to refill its buffers after a play-time change, which is in the order of a few seconds, even the state change of all members of the whole association is conducted in acceptable time.

9.2.2.2 Distinction by Synchronization Mode

For a state change of the association, we also measured the time until the new state is distributed to the users. Here, we run different experiments for independent and reflected synchronization styles. In the independent mode, the association service does not control the sessions of other members on their behalf, but waits for the clients to react on the streamstate notifications. The notifications thus carry a flag containing the synchronization mode. In the reflected mode, clients do not send control requests for an association state change, whereas in the independent mode, each client sends the required control request.

For the following experiments, the costream components as well as the clients reside on the service provider network (*university scenario*). We have captured the reception and sending times on the host running *Cassis*. In order to get the real latency experienced by the clients, the networking delay among *Cassis* and the clients has to be added. However, this delay is in the order of a few milliseconds, because all machines are in the same LAN.

From the results shown in table 9.6, it can be seen that a state change is relatively slow in the independent mode, where the control requests have to be sent by the clients themselves after receiving the streamstate notification. However, the increase in latency is relatively small, from 0.22 s on average for two members until 0.32 s on average for 10 members. Furthermore, the individual clients perceive only the latency that passes for their own control operations, which is in the order of a few milliseconds. The reason for the larger latency is less the association service

Synchronization Mode	Number of Members	min (s)	max (s)	avg (s)	mdev (s)	90th perc. (s)
Reflected	2	0.01	0.29	0.02	0.02	0.03
	3	0.01	0.27	0.01	0.02	0.02
	5	0.01	0.26	0.02	0.02	0.02
	10	0.02	0.24	0.04	0.02	0.06
Independent	2	0.02	0.81	0.22	0.23	0.56
	3	0.02	0.98	0.27	0.28	0.72
	5	0.04	0.92	0.33	0.28	0.72
	10	0.04	0.87	0.32	0.30	0.75

Table 9.6: Latency Measurements for State Change

that has to handle a large number of control requests, but the time required for notification of the members.

The reflected synchronization mode scales better with the number of members for a state change. Up to five members of a group, no difference in the latency can be perceived. Only for 10 members, a higher latency can be perceived, which possibly results from the fact that association service and reflector are separate components and the whole data traffic runs via the *Cassis* host. Furthermore, the implementation of the policy query can be optimized to trigger the state change notification in an asynchronous fashion.

An optimization of the independent mode would be to change the session for the clients which makes them react like in the reflected mode. However, the association service has to maintain a collection of the server sessions of all association members, which is quite costly and considerably reduces the advantage of easy implementation of the independent mode. Moreover, the control requests would still have to be sent for each server session, which requires more time than in the reflected mode, where exactly one control request is sent to the server.

Finally, we compare reflected and independent synchronization mode for the setup of a new association. In this experiment, we have measured the time until the response reached the requester, because the notification of the new state has only informational status for the other members of the association. We therefore also left out the distinction regarding number of members.

Synchronization Mode	min (s)	max (s)	avg (s)	mdev (s)	90th perc. (s)
Reflected	0.02	0.12	0.04	0.02	0.06
Independent	0.01	0.04	0.01	< 0.01	0.02

Table 9.7: Latency Measurements for New Association

In the independent mode, only the session state for exactly one session must be changed. As the results presented in table 9.7 suggest, this leads to very fast control request processing. The reflector introduces some overhead, because the member must leave the reflector session and a new session to the server must be opened. However, this overhead is quite low and does not outperform the advantages regarding scalability and latency that the reflector provides for

managing the streaming sessions of the members of one association. Hence, in all environments where the connection from association service to streaming server is a bottleneck, using the reflector will yield much better latency results than the independent mode does. This is valid for joins to the association as well as for state changes.

9.2.3 Synchronization Evaluation

For the evaluation of synchronization, we take an environment of two clients, which are synchronized to NTP servers to compare the capture times of RTP packets. The accuracy of NTP applied over the global Internet is about 10 ms. Since the standard Linux system (100 Hz) timer resolution is also 10 ms, this is considered enough for our experimental settings.

Only RTP packets with the marker bit of the RTP header set to true are considered for the measurements. Those packets indicate that a full video frame is available for depacketization and decoding. The reception times of other packets do not have to be compared. In order to measure the synchronization deviation, the reception time of the first full video frame delivered to the joining client must be compared with the reception time of the corresponding video frame at the starter client.

In our measurements, only the synchronization deviation among two different clients is measured (inter-client synchronization). For this, we compare the arrival times of corresponding video packets of the streams sent to those clients. The deviation among audio packets is not measured, because RTP audio packets contain several audio samples, which may lead to higher measured deviations of packet reception times than actually existing after depacketization. Moreover, it is not necessary to compare all the streams to evaluate the synchronization mechanisms of our architecture, because RTSP servers are responsible for starting all streams of a track selection from the particular position that has been requested by the clients (or by the association service, in our case). The responsibility of costream's synchronization mechanisms is to choose the correct play-time position for a joining client.

The video track of our example movie is a 320x240 pixel 25 fps MPEG-4 video (Simple Profile Level3b) with a bit rate of 589 kbit/s. The video is packetized according to the MP4V-ES payload of RTP [84]. This means that a video frame may consist of several RTP packets. As already mentioned, the marker bit of the RTP header denotes the last packet of one frame. The selection of codec and packetization style does not have an influence on the costream synchronization.

For each synchronization mode, we have experimented with two scenarios: In the university scenario all components reside in the 1 Gbit/s Ethernet LAN as used in the experiment for latency measurements, whereas the distributed scenario experiments deal with one client in the home access network and another in the LAN where the costream components reside. Both settings are depicted in figure 9.15 for the independent synchronization mode. *Cassis* must calculate the correct position for the joining host. In each scenario, the latency difference between *Cassis* and joining client (PC2) on the one hand and *Cassis* and starting client (PC1) on the other hand plays an important role. In the university scenario, this difference amounts to 0.1 ms, whereas in the home scenario the value of this difference is about 30 ms (measured by ICMP ping round-trip time).

Subsequent experiments have been conducted, once with the joining client in the access network and the other time with the joining client in the remote network. The latter is not shown in the figure, but the client applications can just be exchanged. The distribution of costream components

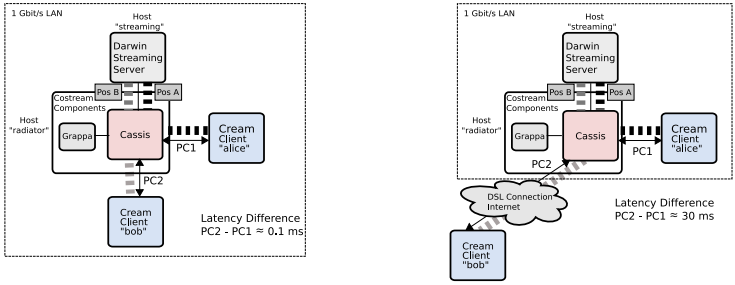


Figure 9.15: University (left) and Home (right) Scenarios in Independent Mode

is not relevant for the synchronization deviation, since they are not part of the media streaming itself. Hence, both the *Grappa* and the *Cassis* application run on one server in the provider network. In the experiments concerning the reflected synchronization mode, the *beaver* reflector application has been added as an additional component on the host *radiator*. This setting is shown in figure 9.16. The *beaver* reflector copies the media packets of the streaming session and sends

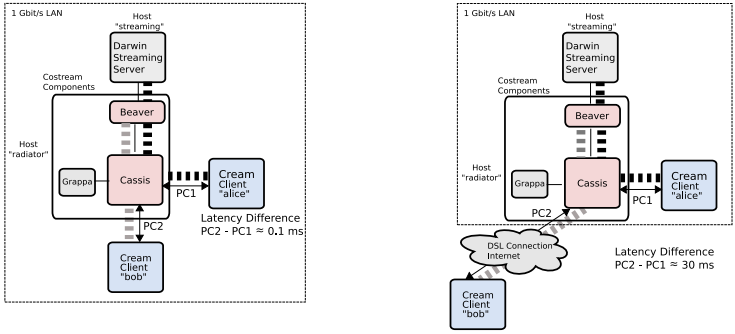


Figure 9.16: University (left) and Home (right) Scenarios in Reflected Mode

them to *Cassis*, which forwards the packets to the clients. As an optimization, both applications could be integrated.

The provider and access networks are switched Ethernet networks. As long as the switch capacity is sufficient, background traffic from and to different hosts does not interfere with the costream applications traffic. At the host radiator as well as on the client hosts themselves, no other applications have been running during our experiments. However, a certain amount of traffic due to multicast announcements which are sent inside the virtual private network (which both provider and access network machines belong to in our test setting) is received during our experiments. We measured this traffic volume to be around 4-7 kbytes per second (depending on the time of day). The traffic volume resulting from running a costream session from provider network to two clients in the access network has been measured as 140 kbps on *radiator* and 75 kbps on the machine both clients were running at. The double amount of data on radiator can be explained

with the fact that this machine is responsible for both receiving and sending media data. In a local area network, jitter appears only rarely. However, the Internet and the DSL connection between access and provider networks are subject to such background traffic, which makes networking jitter more likely.

The experiment to measure synchronization deviation is done by starting the session at the first client and executing a pull from the joining client. The joining client leaves the association afterwards by stopping the streaming session and joins again after 10 s. This is repeated ten times. In each figure, the results of 10 such experiments are collected, i. e. 100 runs. Instead of printing the deviation for every received frame only the first and the 25th frame (roughly corresponding to one second, which is a reasonable value for a jitter buffer) are shown. This is done to examine whether the use of a jitter buffer has effects on the synchronization of clients. For later points in time, it is less important that every single frame is within ideal synchronization bounds, because each client can compensate gaps in packet reception by using the jitter buffer. This holds as long as the value of the network jitter does not exceed the time span of buffered media data and assumed that each client has a clock which is similarly accurate as other clients' clocks. Additionally, the "ideal" lip synchronization bounds of +/- 80 ms are drawn to simplify comparison.

9.2.3.1 Evaluation of Reflected Synchronization

In the first experiment, all components reside in the service provider LAN. The difference of the server-client latencies is very small, about 0.1 ms in absolute terms.

The inter-client synchronization deviation for this scenario is shown in figure 9.17.

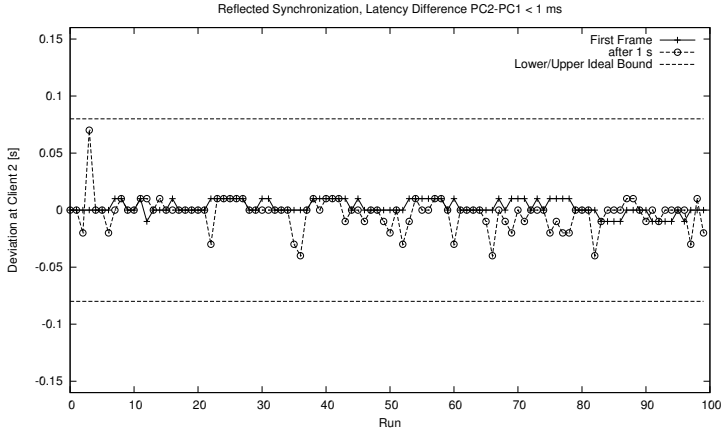


Figure 9.17: Reflected Synchronization for University Scenario

The synchronization of the first frame is mostly good, whereas after the 25th frame a higher deviation can be observed. This results from the fact that the reflector copies packets on the application level. As the following excerpt of a packet capture at the reflector host itself shows,

the reflector happens to send packets of the same frame (indicated by the same packet length) to different clients at different points in time (measured in s):

```
Time  -- Sender    -> Receiver -- Length -- P'col -- SeqNo.
29.73 -- reflector -> bob      -- 630    -- RTP --  Seq=194
29.75 -- reflector -> alice    -- 630    -- RTP --  Seq=582
29.80 -- reflector -> alice    -- 759    -- RTP --  Seq=583
29.81 -- reflector -> bob      -- 759    -- RTP --  Seq=195
29.81 -- reflector -> bob      -- 701    -- RTP --  Seq=196
29.83 -- reflector -> alice    -- 701    -- RTP --  Seq=584
29.87 -- reflector -> alice    -- 610    -- RTP --  Seq=585
29.87 -- reflector -> bob      -- 610    -- RTP --  Seq=197
29.89 -- reflector -> bob      -- 593    -- RTP --  Seq=198
29.91 -- reflector -> alice    -- 593    -- RTP --  Seq=586
29.96 -- reflector -> bob      -- 574    -- RTP --  Seq=199
29.96 -- reflector -> alice    -- 574    -- RTP --  Seq=587
```

In this case, the difference is quite low (between 0 and 20 ms). In a local area network, an optimization could be to use multicast, which would copy packets on the LAN level assumed that the LAN supports multicast. Such an optimization would require changes to both the reflector and the client implementations, however. Since background traffic may introduce jitter among packets even in a local area network, it is nonetheless recommended to apply a jitter buffer. The play-out should start after reception of the first frame plus a certain time, which must be identical on all clients.

In the next experiment, we use a distributed scenario with the starting client in the provider network and the joining client in the access network. The latency difference for a ping from proxy to starting and proxy to joining client is about 30 ms. This means that the joining client receives packets later. In figure 9.18, the synchronization deviation is depicted for the first and the 25th frame.

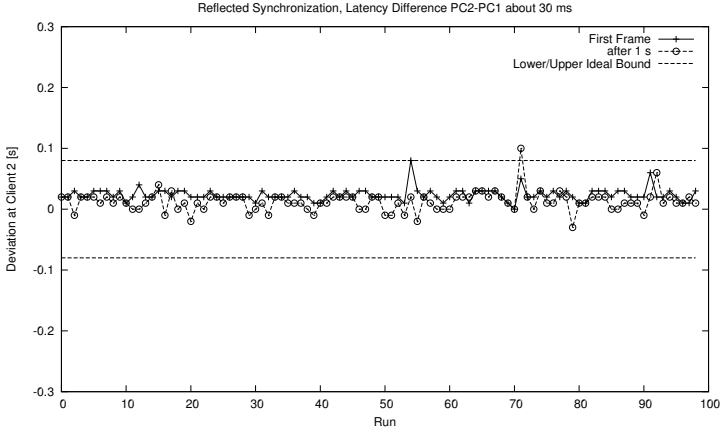


Figure 9.18: Reflected Synchronization for Joining Client in Access Network

On average, the joining client receives the frames about 30 ms later than the starting client. Additionally, the jitter introduced by the network is higher, which also results in higher differences

in the synchronization deviation among clients. However, the deviation stays within the ideal bounds of +/- 80 ms in most cases.

The other case of distributed clients is shown in figure 9.19: The starting client resides in the access network, and another client joins from the service provider LAN. In this case, the deviation

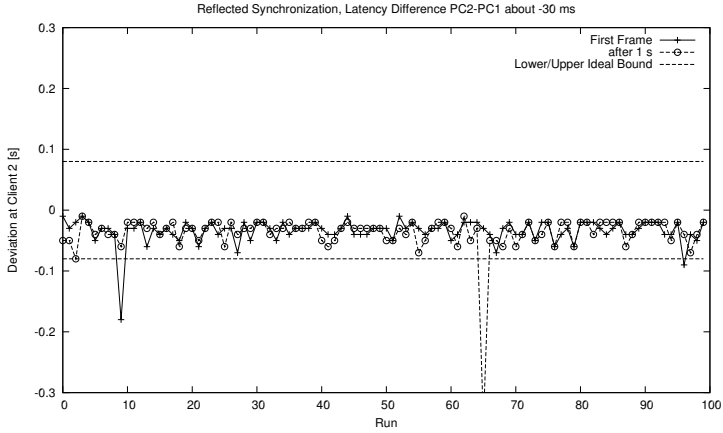


Figure 9.19: Reflected Synchronization for Joining Client in Provider Network

averages at -30 ms, which means that the joining client receives frames earlier. Two runs have a synchronization deviation that exceeds the ideal bounds, resulting from jitter on the DSL connection to the access network. Once, a high deviation of -350 ms could be perceived after one second.

In realistic environments, often both clients are in separate locations in a distributed clients scenario. This means that even higher synchronization deviations as shown here will not matter. Optimizations are not necessary in this case.

However, in case that clients are in the same room it may happen that one client plays out its audio stream which all clients have to listen to. To overcome this deficiency, the clients would have to adapt their play-out buffers according to a synchronized presentation time. If the joining client receives packets later, it has to use a smaller play-out buffer, otherwise it must buffer more frames until the play-out can start. The association service would have to measure the delay to the clients and direct the play-out buffer usage by an extension header during RTSP signaling [161]. Without using clock synchronization, feedback about buffer size and fill level can be given using RTCP, following a similar approach as presented in [147].

9.2.3.2 Evaluation of Independent Synchronization

The independent synchronization mode calculates the play-time position for the client. In the following experiments, the synchronization calculation is done based on local knowledge only: The association service saves start time and position and calculates the time difference if any client joins the association.

Since the delay differences from server to both clients is small in the university scenario, the independent synchronization also provides for good results in many cases, as shown in figure 9.20 with the synchronization deviation at the joining client related to the client that is already a member.

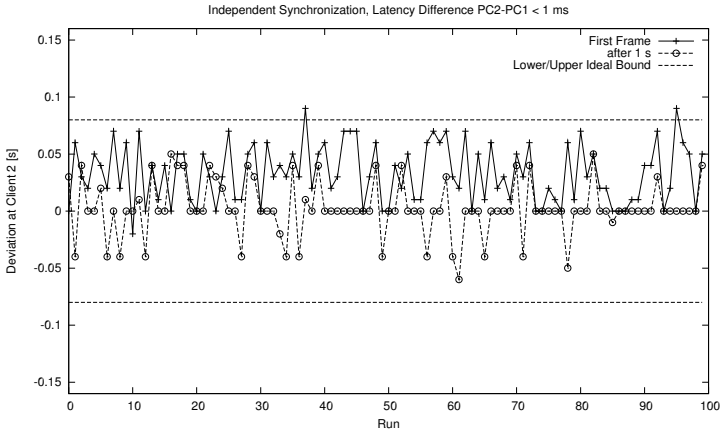


Figure 9.20: Independent Synchronization for University Scenario

For some runs, the synchronization is out of the ideal bounds. Also, the synchronization is not as smooth as in the reflected case, because the synchronization calculation can be wrong even for small changes in the networking conditions. A negative deviation means that the joining client receives the frame earlier than the starting client does, and vice versa for a positive deviation.

In the following experiment, the starting client is in the service provider LAN, whereas the joining client resides in the access network. Figure 9.21 shows the results of the independent synchronization in this case.

It can be seen that sometimes the deviation is outside the ideal bound of +/-80 ms. On average, the joining client receives packets about 80 ms too late. However, for the 25th packet received (corresponding to about one second for a 25 fps video), the deviation is smaller (the value at run 64 is caused by extreme jitter). By counting received packets, we have found out that in many cases, the joining client receives the first few seconds of its streamed video at a slightly higher rate than 25 fps. In about 60 - 70 % of the cases, the rate was greater or equal to 26 fps. Possibly, the streaming server sends the first few seconds at a higher rate.

We also took measurements for the case that the starting client is in the access network and the joining client is in the provider network. Since the joining client experiences a smaller delay to the association server, it receives packets about 70 ms earlier on average, as shown in figure 9.22.

Two measurement values of very large deviation (about -400 ms) after one second can be seen, at run 3 and run 55. In both cases, very large jitter or even packet loss occurred. In this experiment, a similar effect as shown above can be noticed: The joining client receives the 25th frame earlier than the first frame, which however results in higher synchronization deviation.

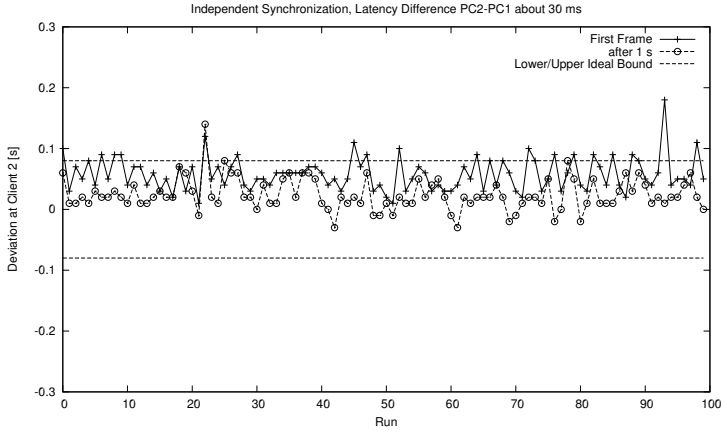


Figure 9.21: Independent Synchronization for Joining Client in Access Network

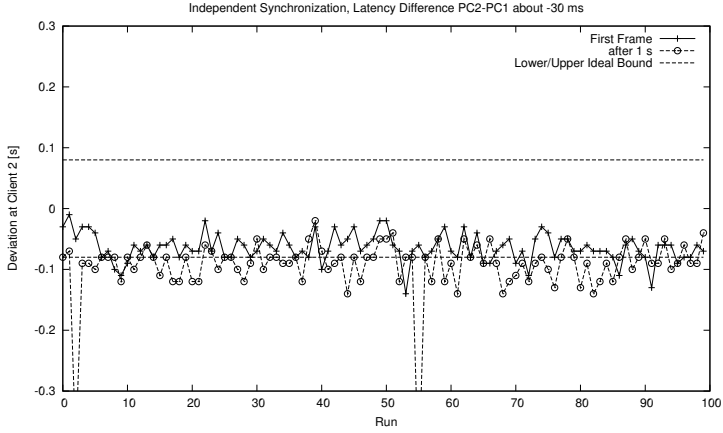


Figure 9.22: Independent Synchronization for Joining Client in Provider Network

The independent synchronization does not provide as good results as the reflected mode. In scenarios with distributed clients, the deviation is higher than for the reflected mode using the same settings.

9.2.3.3 Optimized Synchronization

We have already mentioned in section 8.2 that independent synchronization can be optimized with a delay measurement. The measurement of the proxy-server delay could be taken with a `Timestamp` header on sending a DESCRIBE or SETUP request. However, the Darwin Streaming Server does not echo the Timestamp header, which is not RFC compliant. Hence, the association service has to save the timestamp before sending the request to measure the round-trip delay after reception of the response.

In table 9.8, the request-response latencies of each RTSP method as measured from a host from the home network are summarized. Note that in reality the round-trip delay has to be measured on the proxy host, but in our case the latencies for requests inside the LAN are lower or equal to 10 ms. The delay compensation for improving the independent synchronization mode would work best by measuring the time from sending the PLAY request until receiving the first media data packet. However, at this time the position already has to be calculated. Since the SETUP request has the nearest average value to the PLAY request, a timestamp is saved before sending the SETUP and read after reception of the confirmation response.

RTSP Method	min	max	avg	mdev (s)	median (s)
DESCRIBE	0.04	0.20	0.04	0.02	0.04
SETUP	0.04	0.24	0.05	0.02	0.05
PLAY	0.04	0.22	0.06	0.01	0.06

Table 9.8: Latency Measurements for RTSP Methods

Measuring the networking delay to the client is more difficult, because neither an ICMP ping nor a TCP SYN on the echo port really corresponds to the time it takes to deliver the first RTP packet after a PLAY request from the association service to the client. Nonetheless, we have chosen to take the round-trip time returned by a ping as the client delay value, due to lack of better alternatives.

As shown in figure 9.23, the synchronization deviation at the joining client in relation to the starting client is always within the ideal bounds in this case.

The actual difference among optimized and independent synchronization is virtually imperceptible in this case, though the deviation never exceeds the ideal bounds in this measurement. In such a local environment, the advantage of the optimization is relatively low and a part of the data path is shared among clients, which suggests that the reflected synchronization mode is to be applied instead of any independent mode.

We also have measured the deviation for the case that one client starts from the provider network and the other client joins from the access network. The results of this experiment are shown in figure 9.24.

In this case, the synchronization of the first packet fits within ideal bounds. Again, after the 25th frame, the synchronization deviation is lower for many cases. However, some values at the 25th

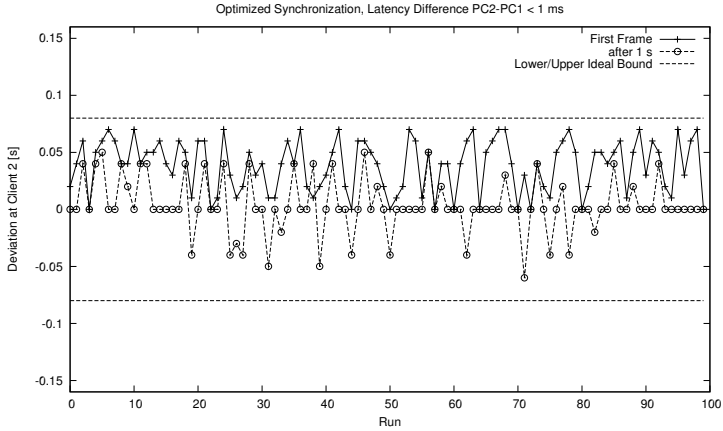


Figure 9.23: Optimized Synchronization for University Scenario

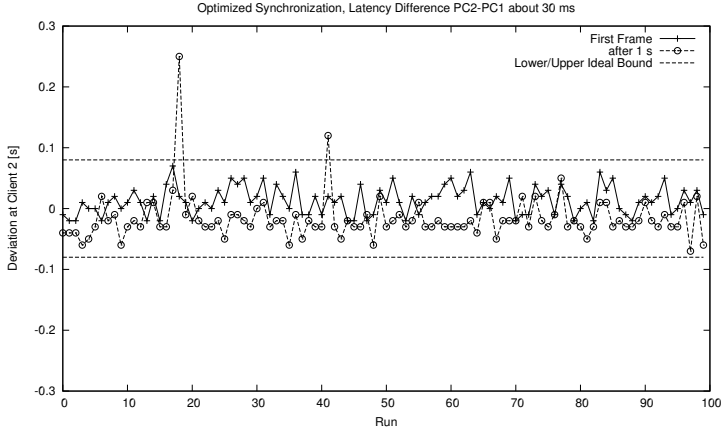


Figure 9.24: Optimized Synchronization for Joining Client in Access Network

frame are out of bounds, possibly because of changing networking conditions or measurement errors due to packet loss.

As shown in figure 9.25 for the starting client in the access network and the joining client in the remote (service provider) network, the delay compensation also in this case provides for a smaller synchronization deviation than the independent synchronization does.

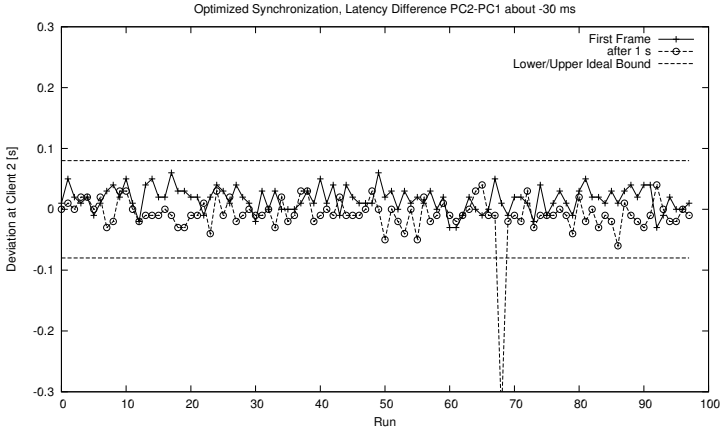


Figure 9.25: Optimized Synchronization for Joining Client in Provider Network

Except for one run in this experiment, where the synchronization after one second is disturbed, the deviation is within the ideal bounds. Similar to the measurements shown in figures 9.22 and 9.19 for independent and reflected synchronization, a jitter event of -310 ms occurred at that time. A more intensive review of the packet captures at the time of such jitter events yields that only the client in the access network has suffered from this jitter event. Moreover, those events occur in case that a video frame consists of many (up to 15) RTP packets. These packets that are sent nearly at the same time may exceed the Internet and/or DSL capacities for a short time-scale. Note that the synchronization of the media data at the clients will normally be better than measured here, because the starting client afflicted with the jitter has filled its play-out buffer already.

Compared with reflected synchronization, the optimized independent synchronization provides for less inter-client synchronization deviation on average. The delay measurements are not very costly and can be integrated into the calculation well. However, the results for reflected synchronization are smoother and therefore more predictable. Varying networking delays to the streaming server or different server response times have no effect on the calculation.

9.2.3.4 Conclusion of Synchronization Measurements

The measurements of the association service synchronization styles show that synchronization is possible to a certain extent, but only on a best-effort base. In scenarios with a low difference of the latencies among association service and clients, good synchronization results can be achieved with reflected synchronization. Another advantages of reflected synchronization are

the transport efficiency due to data path sharing and the faster handling of state changes for the whole association.

For these reasons, independent synchronization should generally only be taken into account if clients require to have an individual connection to the streaming server or if a group plans to split very soon. This is often the case in mobile environments. If independent synchronization without delay measurements is used, clients experience a relatively high synchronization deviation even for low to medium latency differences. The optimized synchronization provides for an effective way to overcome this: in the case of two clients distributed among access and remote network, the inter-client synchronization deviation stays within ideal bounds.

However, it is not possible to guarantee that a calculated delay compensation always works as desired. Some aspects that disturb synchronization even in place of delay compensation are:

- The first synchronization measurement after a longer time had almost always very different characteristics than the following measurements, above all if the starting client experiences a larger delay. The streaming server media delivery takes much longer in this case (e. g. 300 ms vs. 100 ms for following runs). Since this behavior could be perceived also in the case that the streaming server has been run on localhost, it seems to be a server-specific implementation issue. The measured delay to the server cannot compensate for this, because it is based on the response times of RTSP requests.
- Media data packets are vulnerable to jitter caused by other traffic like file transfers or web traffic. Small-scale measurements as done in our architecture cannot fix this. If the change in the network conditions occurs during the delay measurements or during the transmission of the first packets, a good synchronization is hard to achieve. In other cases, the play-out buffer can compensate for jitter.
- Finally, the client machines may have different networking and processing capabilities. Though this has not been measured, we expect this behavior to contribute a lot to synchronization deviation. A particular problem is that different clocks on the systems cause that applications on different hosts run at different speeds. This means that the synchronization deviation can rise during the course of a presentation. One solution to this is to resynchronize during a presentation. Since usual RTSP PLAY requests will be misinterpreted as play-time position control requests, a feedback mechanism using RTCP is a better choice. Another solution to this would be to implement a synchronization controller, which is able to direct player application buffers and hardware device buffers.

Another issue concerning the setting of the play-out buffer can be learnt from the measurements: Only for the case of independent synchronization with the joining client from the access network, the inter-client synchronization deviation has been considerably lower for the 25th frame. Hence, it is recommended that the play-out should not start after a certain number of frames have been received, but after a certain time, which is used for buffering frames at a client and must be identical at all clients. It is reasonable to introduce an RTSP extension header to negotiate such a common play-out buffer time among association service and clients.

Unfortunately, without server-directed presentation timestamps, a synchronization of several clients with quality guarantees is not generally possible. However, for most collaborative streaming scenarios, particularly for synchronization of several video devices or devices in different rooms, the presented synchronization methods are sufficient and prove to be a low-effort alternative to specific synchronization protocols.

9.2.4 Message Overhead of Costream

The communication of costream intermediate components among each other and with servers and clients requires some additional messages. The overhead introduces by these messages is discussed in the following.

The communication among *Cassis* and *Grappa* is done using RMI in our prototype implementation. While it has been shown that this is not disadvantageous concerning latency, some message overhead comes with using this mechanism. We therefore have measured the amount of RMI data exchanged during registration of conferences and participants, and during control operation policy queries.

For the registration of a conference, an RMI connection between *Grappa* and *Cassis* has to be set up. In the following, an overview of the exchanged messages is given, whereas the real packet trace is shown in the appendix. RMI opens a connection to the RMI registry for lookup of a server, and an RMI connection for actual method invocation. For the first method call, this procedure results in 15 RMI messages being exchanged with a summary amount of about 3000 bytes (TCP connection setup and ACKs not counted). In addition to lookup which takes about 790 bytes, a lease with a server identifier is exchanged (1220 bytes) . The liveness of the server JVM is checked with a JRMI Ping to the RMI registry (using 215 bytes) before the actual method invocation is done by a JRMI Call and ReturnData exchange (about 550 bytes in case of conference registration).

Forthcoming method calls for registration of participants only check the server JVM's liveness by a Ping/PingAck exchange before they invoke the method and receive the return data. This processing requires 4 messages and about 690 bytes of data being exchanged. As long the *Grappa* and *Cassis* applications are running, additional conferences can be registered in the same fashion, with a data amount of roughly 790 bytes. For deregistration, the exchanged data average to 390 bytes.

Since both *Grappa* and *Cassis* have to serve and call RMI requests in our prototype implementation, each costream component implements its own RMI server. Hence, for control requests another RMI connection is opened from *Cassis* to *Grappa*. A similar overhead like above has been counted, with 15 messages with 2520 bytes. The policy query uses less data than a conference registration, Lookup requires 791 bytes, lease and UID exchange 1220 bytes, the ping test 215 bytes and the policy query invocation 294 bytes. Again, the connection is re-used for further policy queries, which use 4 messages with about 410 bytes. RMI therefore means a certain message overhead, but has the advantage of simplified development due to the pre-defined protocol for method invocation. Compared to the text-based IETF signaling protocols, RMI requires even lower amounts of data because of the serialization of data.

Regarding RTSP communication, there is not much overhead compared to common RTSP unicast streaming sessions. Only an additional SET_PARAMETER message has to be sent. In our case, this message has a length of about 300 bytes. A poor placement of the *Cassis* RTSP proxy would certainly introduce overhead in parts of the server-client path. However, this is valid for any RTSP/RTP streaming proxy. We also expect providers to couple *Cassis* with reflectors or caches, which makes streaming more efficient. If a reflector is used with *Cassis*, an additional reflector tag header is used, but using reflector sessions saves a lot of bandwidth compared to unicast sessions. In our prototype implementation, we use a separate reflector. The integration of such a

reflector into *Cassis* and the use of multicast in local area networks are further possibilities of optimization.

Finally, the SIP communication of costream is compared to SIP centralized conferencing without streaming sessions. An additional REFER message is required to register the streaming session to the group management. Besides the accepted status response, the REFER requires to notify the initiator about the status of the REFER. If two notification messages (one for trying and one for success / failure status) are assumed, the overall overhead of the streaming session registration is about 2680 bytes. Optimizations could be to suppress REFER notification or to send only the success message, though both of these proposals are disapproved by the SIP standard. However, the registration of the streaming session is normally done only once per collaborative streaming session.

The additional policy information costs about 1000 bytes, which are sent as the body of two messages during the SIP INVITE transaction, following the offer-answer model of the SIP standard. The policy is also sent to other joining clients. Hence, the overhead depends linearly on the number of members. An optimization can be not to inform joining clients about the policy or – in specific scenarios – to use static policies at all.

The additional information about the streaming session in conference event package only requires little additional overhead, about 100-150 bytes. However, the additional streamstate event package means additional SUBSCRIBE / NOTIFY exchanges. For an exemplary subscription to this package, 2150 bytes are required. Subsequent notifications use 1266 bytes, if additional subgroups are used, about 200 bytes have to be added for the subgroup information. These notifications have to be sent at each control request for all members. Hence it is recommended to use partial notifications to keep the notification messages as short as possible.

9.3 Summary

The costream architecture is the first effort to create a collaborative streaming architecture with both synchronized streaming and policy-based control. Furthermore, control operations comprise both the possibility to change the state for all members of a (sub-)group or to change state individually. The concept uses standard IETF signaling protocols. This means that available intermediate components can be used and enhanced, and that the architecture can benefit from a number of functions (e. g. SIP conferencing) that are available in these protocols and increase the value for users.

As discussed in section 9.1, the costream architecture fulfills the demands that have been put on such a comprehensive collaborative streaming architecture. Our prototype works in a number of local and distributed networking scenarios, as we have shown with quantitative results for latency and synchronization.

The latency of initiation transactions introduced by the costream architecture is acceptable for all scenarios. Even the remote access via DSL has proven to be unproblematic, which allows the deployment of costream components in provider networks. In few exceptional cases, latency values rise because of necessary SIP or TCP initialization procedures of the Java implementation. It has to be noted that the human reaction times or the initialization of the media player application take longer time than the measured values. By feedback on the status of their transactions, callers perceive the application as being truly interactive. Hence, the costream architecture proves its effectivity for session transfer operations.

Session control operations can be handled fast; even the query of the group management does not introduce much latency. Session control for an association depends on the synchronization mode also: For a state change of the whole association, the reflected synchronization mode as introduced for multicast-style synchronization proves to be the best choice, because the reflector just forwards the control request to the server. That way, even larger numbers of group members can be handled. The reflector implementation also offers opening of new associations in acceptable time.

The inter-client synchronization with a low or medium delay difference can be done using different synchronization styles. Even a simple calculation of the play-time position can be sufficient in scenarios with low jitter or relaxed synchronization requirements. Such a calculation can be optimized using delay measurements to the streaming server and the clients, which has shown good results even for distributed clients. Finally, a reflector that copies packets and allows for dynamic path reconfiguration provides for good synchronization in local scenarios. This synchronization mode can also be optimized in an easy fashion to deliver common presentation timestamps for all clients. Clearly, modification of the client software is necessary in that case.

Compared to usual Internet streaming and conferencing systems using RTSP or SIP, only the necessary communication among group management and association service and the notification of clients of the collaborative streaming state contribute to considerable message overhead depending on the number of clients and control requests that are issued. However, since the communication among costream components is implemented using RMI and partial notifications can be used for streamstate events, even these additional messages do not use bandwidth excessively. Compared to the bandwidth usage of the media packets, the additional overhead is very low. Since standard IETF protocols are applied, provisions for optimization of bandwidth usage can be applied to our architecture also, e. g. SIP header compression for mobile scenarios.

All in all, the concept of a collaborative streaming architecture using IETF protocols proves to be applicable in a number of scenarios. Due to flexible group policies, users receive the additional benefit of controlling the collaborative session in a defined fashion. Furthermore, session transfer primitives, implemented by initiation transactions, allow to easily incorporate other users or even devices into a streaming session.

10. Conclusion and Outlook

The increase of network bandwidth on the one hand and the development of consumer electronics devices with network access on the other hand have increased the interest for multimedia services using Internet technology. People are interested to combine video on-demand services with telephone services to call others so as to jointly watch streaming presentations. At the same time, the control functionality known by VCRs or DVD players must be retained to avoid restrictions of the service. Such a collaborative streaming service is also interesting for learning scenarios, where students elaborate topics moderated by a supervisor.

The range of functions of a collaborative streaming architecture comprises *session sharing* on the one hand, which means the provision of a synchronized streaming service to the participants of a group, and the possibility to influence the course of a presentation on the other hand. This *session control* must result in a deterministic behavior in accordance with group policies. Moreover, *session transfer* functionality must be available for clients so that they can pull a streaming session from others or push it to others.

In this thesis, a collaborative streaming architecture capable of handling both a common group state and individual session control operations has been designed, implemented and tested extensively. Since the approach presented in this work is one of the first comprehensive collaborative streaming architectures, scenarios and preconditions for collaborative streaming have been examined in chapter 2. It has been found that home, learning and spontaneous meeting environments provide enough common functionality to offer a common collaborative service. However, individual collaborative groups inside those scenarios have different interests during the course of the presentation. Hence, the notion of a collaborative streaming group, consisting basically of its management state and its streaming session state, has been defined in chapter 4. This definition led to a separation of those parts of the session state that are controlled in a shared fashion and the individually changeable variables. In order to enable a flexible shared control and conflict resolution, the notion of an association as a subgroup in which the members have an identical shared session state has been introduced.

The high-level basic functions as they are perceived by users have to be mapped on the collaborative streaming service. Since group management and session control functionality should be kept separate, we developed a collaborative streaming service as a composition of group management and association service, which have been shown in chapter 5 in greater detail. Both services provide for a number of functions which implement session transfer and session control in a flexible and extensible fashion.

The realization of our collaborative streaming service using IETF standard protocols has been presented in chapter 7. A mapping of the mentioned functions to SIP and RTSP control requests

has been developed, and the costream architecture has been designed and implemented enhancing standard signaling components.

10.1 Results

The costream architecture provides for a flexible streaming service for groups with a tight relationship. It is implemented using IETF multimedia signaling protocols. Therefore, it can be deployed in all networking environments that support the Internet protocol suite, which makes costream applicable nearly everywhere.

Costream follows a centralized group concept, because collaborative streaming groups are coupled more tightly than groups of multicast streaming session receivers. One part of the group concept is a common group policy that governs the execution of membership and session control operations. Inside this policy, the piece of dynamic session state (e. g. play-time position, or tracks) that is managed jointly for the group members is defined. Users are given certain roles within a collaborative streaming group. Each role has a particular set of permissions, which allows to implement access control in accordance to the specific permissions. For membership control, the permissions are to join a group and/or to push the session to another client, whereas the permissions for session control consist of pausing and/or changing the play-time. A difference to usual access control models is the definition of a so-called reaction policy, which determines whether an association has to change state or has to be partitioned. The particular implementation of the group policy is not dictated by this approach, Since groups may have very different interests, each administrator or creator of a group defines own policies compliant to the above-mentioned model.

The costream architecture separates group management from the management of streaming session state. The first is done using SIP conferencing functionality, whereas the latter applies standard RTSP control requests to a subgrouping concept. Groups can be split into so-called associations which represent a common piece of dynamic session state inside a group. The separation of group and session management allows a streaming session to be started before or after group setup, which makes the approach applicable in a range of scenarios, even for spontaneous collaborative sessions.

Costream provides for synchronization of streaming sessions without having to use multicast transport or global clock synchronization. The synchronization using a reflector works similar to multicast, whereas the so-called independent synchronization allows to build separate streaming sessions for each client. This is useful for groups in mobile scenarios, where each client accesses the streaming service of its own provider and providers do not allow sharing of the data path.

In order to implement the above-mentioned basic functions of collaborative streaming, the following architectural components have been contributed:

Association Service which provides for synchronization to a common play-time position for all registered members of an association. Moreover, for each control request the policy decision about admittance and reaction is queried from the group management. According to this decision, the control operation is denied, the state for the whole association is changed, or a new association is opened. All in all, the association service provides for the management of streaming sessions on behalf of collaborative group participants.

Group Management which implements a number of session transfer primitives to initiate a collaborative streaming session. Whereas Start or StartGroup define a new collaborative group, Push or Pull transactions allow to add members to an existing collaborative group. Since the necessary message flows for these initiation transactions are based on SIP transactions for conferencing, the group management is implemented on top of a SIP conferencing server, which includes a conference factory and a number of conference foci.

Collaborative Client This component serves as the access point for users to the collaborative streaming service. Therefore, all control functionality is implemented on client side. A separation between initiation (session transfer) and update (session control) operations allows for the separation of the client into a SIP and an RTSP part. The application is designed in a way that allows exchange of the player application, the GUI toolkit, and the SIP stack with as little modification to existing code as possible.

The evaluation of the system has shown that the introduced latency due to the combination of SIP and RTSP remains in an admissible range even for home scenarios where a home network has an ordinary DSL connection to the Internet. Push and pull functionality can be executed by means of SIP standard conferencing methods. In case a reaction of a human user is required, the reaction time of this user is always larger than the latency introduced by our architecture. By using SIP, clients receive information about the state of their own requests and of the whole group. Hence, the users perceive a truly collaborative application. Session control methods are standard RTSP methods, which allows the easy integration of standard streaming servers. The latency for an update of a collaborative streaming group by these session control operations is also acceptable, even in those cases the group policy decision must be retrieved by the group management. In our prototype implementation, only the play-time position has been managed in a shared fashion. The communication among group management and association service has been realized using RPCs and shows small latencies also, which allows to deploy those costream components on different hosts.

The message overhead of costream is also in an acceptable range, compared to usual streaming and conferencing services. Only the notifications of the streaming session state require considerable resources, because they are sent to each group participant. This can be mitigated by using partial notifications or sending notifications only within associations. Since the media data transport is most costly within collaborative streaming, the reflected synchronization mode should be used to provide for a scalable service, because all streaming sessions of one association can be subsumed this way.

The synchronization has proved acceptable results even without clock synchronization. For low delay differences up to +/- 80 ms, the reflected synchronization mode can provide for synchronization within ideal bounds. The independent synchronization mode is best used together with a delay compensation, which allows to overcome networking delays. Both costream synchronization modes can be combined with feedback-based synchronization or specific synchronization protocols, which use clock-synchronization to achieve guaranteed bounded synchronization deviations.

10.2 Future Work

The costream prototype implementation has proved to be an effective collaborative streaming architecture. Of course, there is room for improvements of the prototype implementation. Besides

mere performance optimizations, the user-friendliness of our client application can be improved by programming user interfaces tailored to the needs of humans, particularly with guidance regarding configuration of collaborative group policies. The format and processing of these policies will also be enhanced, also regarding the standardization efforts of the SIP and XCON working groups of the IETF. Another interesting future enhancement is the implementation of tracks as a shared state attribute.

The architectural concept of costream can be enhanced in some aspects, which is simplified by the separation of group and session management. As already mentioned, synchronization with quality guarantees can only be achieved using a global clock. A future enhancement would then be to distribute global presentation timestamps to clients or to enhance costream by a synchronization protocol like ASP [139] or a controller mechanism as used in NMM [95].

In addition to those implementation optimizations, which do not require any changes to the general costream architecture, some interesting aspects with respect to research are summarized in the following.

10.2.1 Fault Tolerance

Though the costream architecture works with centralized intermediate components, a certain level of failure tolerance can be achieved. Each component can be replicated, but group members must use a common group management and association service in the concept presented in this thesis. Except for a common registration policy, no global knowledge has to be shared among replicated components, but states can be managed on a per-group basis.

The Grappa group manager can be implemented on client systems also, since the underlying conference focus is a SIP User Agent. However, provisions have to be made to be able to register the streaming session to the association service. Since the association service has to provide for a common synchronization state, it is difficult to be replicated. However, the starting time and position of a certain association may be saved on a back-up system, which may be used for recovery after failure.

A further topic would be is the recovery of group state from failures. Since conference and streaming session state is distributed to clients using notifications, each client itself has a certain knowledge of group and session state. In case of failure, this knowledge can be used to recover the group, possibly connecting to a different group manager or association service, which would have to support such recovery behavior. How far such recovery behavior can be run automatically for whole groups or individual clients is an open topic for future research.

10.2.2 Transport Efficiency

In this work, transport efficiency has not been a main concern, though the reflected synchronization mode reduces the data amount that is sent from the server to the reflector. Moreover, existing proposals made to enhance transport efficiency in standard streaming architectures can be integrated with the costream architecture. First, caching can be used to serve subsequent requests for one piece of media, which occurs in costream if clients change the play-time position during the course of a presentation. Second, clients with different device capabilities can be served using differently coded media files or transcoding mechanisms. The signaling must then be improved by signaling meta-data like device characteristics to choose an optimal data format.

In scenarios using LAN environments, particularly in combination with the reflected synchronization mode, multicast transport is an efficient data transport alternative. The RTSP client implementation must then be modified to implement a change of the play-time position as a change of the multicast group. Since the costream architecture generally allows to use different streaming servers as long as a common association service is used, other approaches like content distribution networks can also be combined with costream. Interesting questions in this context are how synchronization can be handled and in which case additional costream components become necessary.

10.2.3 Mobility

All costream scenarios can benefit from mobility support. In future, more and more multimedia services will be offered for mobile networks and devices. People will watch movies on their mobile devices. Thus, a streaming architecture should support handover of a streaming session to the home network. Middleware architectures like NMM [94] partly support such session handover. In learning scenarios, mobile learners connect to the lectures of a learning center.

An interesting feature of the IETF application layer signaling protocols is their inherent support of mobility. As already stated in 6.1 and 6.2.2, respectively, the signaling protocols need no persistent transport connection. Even without the support of a network-layer mobility protocol like Mobile IP, SIP supports different mobility scenarios, because clients can send re-INVITE requests with updated network information. In RTSP, transport information can be updated using SETUP requests.

In the costream architecture, session mobility is already addressed by implementing the move initiation service. Handovers from mobile devices to home devices can also be implemented that way, provided that a group management and an association service can be connected from both mobile and home network and the above-mentioned requests are used to update networking information. Even if a handover of the streaming session cannot be done seamlessly (e. g. because the streaming server does not support this), the client can join the association again using a new streaming session to the server, because the flexible internal state management of the association allows closing a streaming session without de-registration.

In case that different group management and association service applications are implemented for the particular mobile and home networks, the question comes up in which cases and how a handover from costream intermediate components on one network to those on the other network must be executed. A similar case is also imaginable for spontaneous meetings, in case that two users are attached to different costream components but want to have a common collaborative streaming session. In both cases, the costream components must cooperate, either for handover of a collaborative streaming session or to manage collaborative group state in a distributed fashion. The necessary enhancements of the costream architecture to provide for this are an open topic to examine.

A. Streamstate Event Package Definition

In this chapter, the definitions for the streamstate event package are contained. Due to lack of space, this definition is not compliant to standard rules for event package definition, but may serve as a starting point for such a definition. According to the SIP standard for event notification [130], new event packages must be registered with the IANA.

In the streamstate event package, the dynamic session state of a collaborative streaming session is delivered. It contains the tracks and the play-time position, tagged with a timestamp, for all associations of a collaborative streaming group. The package can be used by a focus acting as a collaborative streaming group manager on behalf of an association service component. The package defines the following event header: `event: streamstate`. The format of the notification state is `application/x-streamstateinfo+xml`, which will be explained below.

A.1 Streamstate Document Format

Though no formal registration of the streamstate event package is done, streamstate elements are associated with the XML namespace name `urn:ietf:params:xml:ns:xstreamstateinfo`, similarly to the namespace used in the conference event package.

The root of an `application/xstreamstateinfo+xml` object is a `<streamstate-info>` element. The `<streamstate>` element must have an `'entity'` attribute. The value of the entity attribute is the SIP URI of the user agent publishing this streamstate document. A `'version'` attribute carries a version number for each subscriber.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="urn:ietf:params:xml:ns:x-streamstate"
   xmlns:tns="urn:ietf:params:xml:ns:x-streamstate"
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
   elementFormDefault="qualified"
   attributeFormDefault="unqualified">

 <!-- This import brings in the XML language attribute xml:lang-->
 <xs:import namespace="http://www.w3.org/XML/1998/namespace"
 schemaLocation="http://www.w3.org/2001/xml.xsd"/>

 <xs:element name="streamstate-info" type="streamstate-type"/>

 <xs:complexType name="streamstate-type">
    <xs:sequence>
     <xs:element name="uri" type="xs:anyUri" use="required"/>
     <xs:element name="playout-state" type="playstate-type"
         minOccurs="0"/>
   <xs:element name="playtime-pos" type="playtime-type"
         minOccurs="0"/>
   <xs:element name="tracks" type="tracks-type"
         minOccurs="0"/>
```

```
  <xs:element name="subgroups-streamstate" type="substate-type"
        minOccurs="0"/>
    </xs:sequence>
   <xs:attribute name="entity" type="xs:anyURI" use="required"/>
   <xs:attribute name="state" type="state-type" use="required"/>
   <xs:attribute name="version" type="xs:unsignedInt" use="required"/>
</xs:complexType>

<xs:simpleType name="state-type">
 <xs:restriction base="xs:string">
   <xs:enumeration value="full"/>
   <xs:enumeration value="partial"/>
   <xs:enumeration value="deleted"/>
 </xs:restriction>
</xs:simpleType>

<!--
    Playout-State
-->
<xs:simpleType name="playstate-type">
  <xs:restriction base="xs:string">
    <xs:enumeration value="playing"/>
    <xs:enumeration value="paused"/>
    <xs:enumeration value="stopped"/>
  </xs:restriction>
</xs:simpleType>


<!--
    Playtime Position
-->
<xs:complexType name="playtime-type">
  <xs:sequence>
    <xs:element name="playtime" type="xs:float" use="required"/>
  </xs:sequence>
  <xs:attribute name="timestamp" type="xs:dateTime" use="required"/>
</xs:complexType>


<!--
    Tracks
-->
<xs:complexType name="tracks-type">
  <xs:sequence>
    <xs:element name="entry" type="xs:string"
       maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>


<!--
    Subgroups
-->
<xs:complexType name="substate-type">
  <xs:sequence>
     <xs:element name="entry" type="subentry-type"
        maxOccurs="unbounded" />
  </xs:sequence>
  <xs:attribute name="state" type="xs:string" use="required" />
</xs:complexType>

<xs:complexType name="subentry-type">
   <xs:sequence>
    <xs:element name="uri" type="xs:anyUri" use="required"/>
    <xs:element name="playout-state" type="playstate-type"
        minOccurs="0"/>
    <xs:element name="playtime-pos" type="playtime-type"
        minOccurs="0"/>
    <xs:element name="tracks" type="tracks-type"
        minOccurs="0"/>
   </xs:sequence>
</xs:complexType>
```

B. Policy Definition

In the following subsections, the definition of a policy document following the XML schema style is given. For illustration, we provide for an example of the application in a configuration file. The same document can also be sent as message body to a SIP INVITE request, with a content-type of application/policy+xml.

B.1 Policy Document

The XML policy document has been kept simple. In the Internet draft of the XCON working group [114], a data model with more complex member and permission entries has been proposed. The policy definition shown here may be adapted to this model, or the XCON model may used for the member policy.

In this policy definition, role names are not subject to restrictions, though it is assumed that XCON roles may be used. For the costream shared attributes, entry values of `playtime` and `tracks` have been assumed. Values for the actions can be `Pause`, `ChangePlaytime`, and `ChangeTracks`, whereas for the reactions `ChangeState` and `ChangeAssociation` are valid values.

For the occurrence of elements, as little restrictions as possible have been made. Instead, implementations may use defaults for member or permission entries in case that none are given in the policy document.

```
<xs:element name="policies" type="policies-type"/>

<xs:complexType name="policies-type">
   <xs:sequence>
    <xs:element name="member-policy" type="member-policy-type" minOccurs="0"/>
    <xs:element name="costream-policy" type="costream-policy-type" minOccurs="0"/>
</xs:complexType>

<!--
    Member-Policy
-->
<xs:complexType name="member-policy-type">
 <xs:sequence>
    <xs:element name="max-members" type="xs:unsignedInt" minOccurs="0"/>
    <xs:element name="roles" type="roles-type" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>

<!--
    Costream-Policy
-->
<xs:complexType name="costream-policy-type">
  <xs:sequence>
    <xs:element name="shared" type="shared-type" minOccurs="0"/>
```

```
    <xs:element name="roles" type="roles-type" minOccurs="0"/>
    <xs:element name="transactions" type="transactions-type" minOccurs="0"/>
  </xs:sequence>
</xs:simpleType>

<!--
    Shared Attributes
-->
<xs:complexType name="shared-type">
  <xs:sequence>
    <xs:element name="entry" type="entry-type"
       maxOccurs="2"/>
  </xs:sequence>
</xs:complexType>

<xs:simpleType name="entry-type">
   <xs:restriction base="xs:string">
      <xs:enumeration value="playtime"/>
      <xs:enumeration value="tracks"/>
   </xs:restriction>
</xs:simpleType>

<!--
    Roles
-->
<xs:complexType name="roles-type">
  <xs:sequence>
    <xs:element name="role" type="role-type"
       maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="role-type">
  <xs:sequence>
    <xs:element name="members" type="members-type" use="required"/>
    <xs:element name="permissions" type="permissions-type" use="required"/>
  </xs:sequence>
  <xs:attribute name="name" type="xs:string" use="required"/>
</xs:complexType>

<!--
    Members
-->
<xs:complexType name="members-type">
  <xs:sequence>
    <xs:element name="member" type="xs:string"
       maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<!--
    Permissions
-->
<xs:complexType name="permissions-type">
  <xs:sequence>
    <xs:element name="permission" type="xs:string"
       maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<!--
    Transactions
-->
<xs:complexType name="transactions-type">
  <xs:sequence>
    <xs:element name="transaction" type="transaction-type"
       maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="transaction-type">
  <xs:sequence>
```

```
     <xs:element name="action" type="action-type"
        use="required"/>
     <xs:element name="reaction" type="reaction-type"
        use="required"/>
   </xs:sequence>
 </xs:complexType>

 <xs:simpleType name="action-type">
    <xs:restriction base="xs:string">
       <xs:enumeration value="Pause"/>
       <xs:enumeration value="ChangePlaytime"/>
       <xs:enumeration value="ChangeTracks"/>
    </xs:restriction>
 </xs:simpleType>

 <xs:simpleType name="reaction-type">
    <xs:restriction base="xs:string">
       <xs:enumeration value="ChangeState"/>
       <xs:enumeration value="ChangeAssociation"/>
    </xs:restriction>
 </xs:simpleType>
```

B.2 Example

In the example policy document presented in the following, directions for processing membership control requests like joining a group or inviting others to a group are given in the `<member-policy>` element. In this example, clients coming from the `foo.com` domain are *observers* only, which are allowed to join the collaborative streaming group, but not allowed to push the session to anybody. Clients from `example.com` are both allowed to join and to invite others, whereas clients from other domains may not join the collaborative group at all.

The values given in the `<costream-policy>` element make provisions for streaming session control operations. In this case, only the play-time position is indicated as shared state. That means, requests for changing the track selection are not subject to the group policy. The policy decision does not have to be queried by the association service for such requests. In this example, we also have subsumed the *pause* and *changePlaytime* operations under a common *controller* role. The user *alice* of the `example.com` domain is the only one to control the streaming presentation in the case of this example. Furthermore, a *pause* action always changes the state for the whole group, whereas a *changePlaytime* would set the controller (here: user alice) to a new association.

```
<?xml version='1.0' encoding='UTF-8'?>
<policies>
    <member-policy>
        <max-members>15</max-members>
        <roles>
            <role name="OBSERVER">
                <members>
                    <member>*foo.com</member>
                </members>
                <permissions>
                    <permission>JOIN</permission>
                </permissions>
            </role>
            <role name="PARTICIPANT">
                <members>
                    <member>*example.com</member>
```

```
                </members>
                <permissions>
                    <permission>JOIN</permission>
                    <permission>INVITE</permission>
                </permissions>
            </role>
        </roles>
    </member-policy>
    <costream-policy>
        <shared>
            <entry>playtime</entry>
        </shared>
        <roles>
            <role name="CONTROLLER">
                <members>
                    <member>sip:alice@*example.com</member>
                </members>
                <permissions>
                    <permission>Pause</permission>
                    <permission>ChangePlaytime</permission>
                </permissions>
            </role>
        </roles>
        <transactions>
            <transaction>
                 <action>Pause</action>
                 <reaction>ChangeState</reaction>
            </transaction>
            <transaction>
                 <action>ChangePlaytime</action>
                 <reaction>ChangeAssociation</reaction>
            </transaction>
        </transactions>
     </costream-policy>
</policies>
```

Acronyms

3GPP	3rd Generation Partnership Project
AAC	Advanced Audio Coding
ADSL	Asymmetric Digital Subscriber Line
AOR	Address Of Record
API	Application Programming Interface
APP	Application Packet (RTCP)
ASP	Advanced Synchronization Protocol
AV	Audio Video
BFCP	Binary Floor Control Protocol
Cassis	Collaborative Association Service
CC/PP	Composite Capability / Preference Profiles
CDN	Content Distribution Network
CNAME	Canonical Name (RTCP)
CORBA	Common Object Request Broker Architecture
CPCS	Collaborative Playback Control Server (Comodin)
CPE	Customer Premise Equipment
CPSM	Collaborative Playback Session Manager (Comodin)
Cream	Collaborative Streaming Client
CRLF	Carriage Return Line Feed
CSCW	Computer Supported Cooperative (or Collaborative) Work
CSMA/CD	Carrier Sense Multiple Access with Collision Detection
DA	Directory Agent (SLP)
DDL	Data Definition Language
DNS	Domain Name Service
DSL	Digital Subscriber Line
DVD	Digital Versatile Disc (formerly Digital Video Disc)
EP	Endpoint
FTTC	Fiber to the Curve
GNU	GNU's Not Unix
Grappa	Group Management Application
GUI	Graphical User Interface
H.323	Video-conferencing over LAN Recommendation of ITU-T
HAVi	Home Audio Video Interoperability
HDTV	High Definition TeleVision
HSDPA	High Speed Download Packet Access
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
HWM	High Water Mark (ASP)
ICMP	Internet Control Message Protocol
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IMS	IP Multimedia Subsystem
IP	Internet Protocol
IPSec	Internet Protocol Security

ISDN	Integrated Services Digital Network
ITU	International Telecommunication Union
ITU-T	ITU - Telecommunication Standardization Sector
JMF	Java Media Framework
LAN	Local Area Network
LDAP	Lightweight Directory Access Protocol
LTB	Lower Target Bound (ASP)
LWM	Low Water Mark (ASP)
MAC	Medium Access Control
MBone	Multicast Backbone
MC	Multiport Controller (H.323)
MCU	Multipoint Control Unit (H.323)
MHP	Multimedia Home Platform
MIME	Multipurpose Internet Mail Extensions
MMUSIC	Multiparty Multimedia Session Control
MP	Multiport Processor (H.323)
MP4	MPEG-4 Container Format
MPEG	Motion Picture Experts Group
MPEG-4	AV Coding Standard of MPEG
NIST	National Institute for Standards and Technology
NMM	Network-Integrated Multimedia Middleware
NPT	Normal Play Time
NTP	Network Time Protocol
OWL-S	Web Ontology Language for Services
P2P	Peer-to-Peer
PDA	Personal Digital Assistant
PSTN	Public Switched Telephone Network
PVR	Private Video Recorder
QoS	Quality of Services
RBAC	Role Based Access Control
RDF	Resource Description Framework
RDT	Real Data Transport
RFC	Request For Comments
RMI	Remote Method Invocation
RR	Receiver Report (RTCP)
RTCP	RTP Control Protocol
RTP	Real-Time Transport Protocol
RTP/I	RTP with Interactive Extension
RTSP	Real Time Streaming Protocol
SA	Service Agent
SDP	Session Description Protocol or (Bluetooth) Service Discovery Protocol
SH	Stream Handler
SIP	Session Initiation Protocol
SLP	Service Location Protocol
SMIL	Synchronized Multimedia Integration Language
SMPTE	Society of Motion Picture and Television Engineers
SOAP	Simple Object Access Protocol
SR	Sender Report (RTCP)
STB	Set Top Box
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TU	Technische Universität
TV	Television
UA	User Agent (SIP or SLP)
UAC	User Agent Client
UAS	User Agent Server

UDDI	Universal Description, Discovery and Integration
UDP	User Datagram Protocol
UI	User Interface
UMTS	Universal Mobile Telecommunications System
UPnP	Universal Plug and Play
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
USA	United States of America
USB	Universal Serial Bus
UTB	Upper Target Bound (ASP)
VCR	Videocassette Recorder
VDSL	Very High Speed Digital Subscriber Line
VLC	VideoLAN Client
VoD	Video on Demand
WG	Working Group (of the IETF)
WLAN	Wireless Local Area Network
WWICE	Window to the World of Information, Communication and Entertainment
XCON	Centralized Conferencing
XML	eXtended Markup Language

Bibliography

[1] A. Adams, J. Nicholas, and W. Siadak. Protocol Independent Multicast - Dense Mode (PIM-DM): Protocol Specification (Revised). RFC 3973, The Internet Engineering Task Force, January 2005.

[2] Akamai Technologies, Inc. Akamai Media Delivery. `http://www.akamai.com`.

[3] Apple Computer, Inc. Mac OS X Server QuickTime Streaming Server 5.5 Administration. `http://images.apple.com/server/pdfs/QT_Streaming_Server_v10.4.pdf`, April 2005.

[4] Apple Computer, Inc. QuickTime Overview. `http://developer.apple.com/documentation/QuickTime/RM/Fundamentals/QTOverview/QTOverview.pdf`, August 2005.

[5] A. Awe, J. Boston, M. Kim, B. Luken, E. So, P. Westerink, and S. Wood. IBM Toolkit for MPEG-4, version 1.3.0. `http://www.alphaworks.ibm.com/tech/tk4mpeg4`, December 2006.

[6] H. Baldus, M. Baumeister, H. Eggenhuisen, A. Montvay, and W. Stut. WWICE: an architecture for in-home digital networks. In K. Nahrstedt and W. Feng, editors, *Multimedia Computing and Networking 2000*, pages 196–203, December 1999.

[7] M. Barnes, C. Boulton, and O. Levin. A Framework for Centralized Conferencing. RFC 5239, The Internet Engineering Task Force XCON WG, June 2008. Internet Draft, Work in Progress.

[8] Beaver – RTSP/RTP Proxy Implementation. Project Page, `http://www.ibr.cs.tu-bs.de/projects/beaver/`, July 2006.

[9] BITKOM – Bundesverband Informationswirtschaft, Telekommunikation und neue Medien e.V. Daten zur Informationsgesellschaft – Status Quo und Perspektiven Deutschlands im internationalen Vergleich. Berlin, 2004.

[10] A. Black, J. Huang, R. Koster, J. Walpole, and C. Pu. Infopipes: An abstraction for multimedia streaming. *Multimedia Systems (special issue on Multimedia Middleware)*, 8(5):406–419, December 2002. ACM / Springer Verlag.

[11] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services. RFC 2475, The Internet Engineering Task Force, December 1998.

[12] R. Braden, D. Clark, and S. Shenker. Integrated Services in the Internet Architecture: an Overview. RFC 1633, The Internet Engineering Task Force, June 1994.

[13] J. Brandt and L. Wolf. A Gateway Architecture for Mobile Multimedia Streaming. In *European Symposium on Mobile Media Delivery (EuMob06)*, Alghero, Italy, September 2006.

[14] J. Brandt and L. Wolf. Multidimensional Transcoding for Adaptive Video Streaming. In *Proceedings of the 17th International workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'07)*, Urbana, Illinois, June 2007.

[15] A. J. Cahill and C. J. Sreenan. An Efficient Resource Management System for a Streaming Media Distribution Network. *International Journal of Interactive Technology and Smart Education (ITSE)*, 3(1):31–44, February 2006.

[16] B. Cain, S. Deering, I. Kouvelas, B. Fenner, and A. Thyagarajan. Internet Group Management Protocol, Version 3. RFC 3376, The Internet Engineering Task Force, October 2002.

[17] G. Camarillo. Message Body Handling in the Session Initiation Protocol (SIP). draft-camarillo-sip-body-handling-03.txt, The Internet Engineering Task Force SIP WG, August 2008. Internet Draft, Work in Progress.

[18] G. Camarillo, G. Eriksson, J. Holler, and H. Schulzrinne. Grouping of Media Lines in the Session Description Protocol (SDP). RFC 3388, The Internet Engineering Task Force, December 2002.

[19] G. Camarillo, A. Niemi, M. Isomaki, M. Garcia-Martin, and H. Khartabil. Refering to Multiple Resources in the Session Initiation Protocol (SIP). draft-ietf-sip-multiple-refer-03.txt, The Internet Engineering Task Force SIP WG, December 2007. Internet Draft, Work in Progress.

[20] G. Camarillo, J. Ott, and K. Drage. The Binary Floor Control Protocol (BFCP). RFC 4582, The Internet Engineering Task Force, November 2006.

[21] S. Casner. Session Description Protocol (SDP) Bandwidth Modifiers for RTP Control Protocol (RTCP) Bandwidth. RFC 3556, The Internet Engineering Task Force, July 2003.

[22] Y.-H. Chu, S. G. Rao, S. Seshan, and H. Zhang. Enabling Conferencing Applications on the Interent using an Overlay Multicast Architecture. In *SIGCOMM'01*, pages 55–67, San Diego, California, USA, August 2001. ACM.

[23] J. Clark and S. DeRose (editors). XML Path Language (XPath) Version 1.0. W3C Recommendation `http://www.w3.org/TR/xpath`, November 1999.

[24] L. Clement, A. Hately, C. von Riegen, and T. Rogers (editors). UDDI Version 3.0.2. UDDI Spec Technical Committee Draft, `http://uddi.org/pubs/uddi_v3.htm`, October 2004.

[25] Arcor AG & Co.KG. Arcor - Video on Demand. `http://www.arcor.de/vod/vod_1_0.jsp`, 2007.

[26] C. D. Cranor, M. Green, C. Kalmanek, D. Shur, S. Sibal, J. E. van der Merwe, and C. J. Sreenan. Enhanced Streaming Services in a Content Distribution Network. *IEEE Internet Computing*, 05(4):66–75, July/August 2001.

[27] D. Beckett (editor). RDF/XML Syntax Specification (Revised). W3C Recommendation, February 2004.

[28] T-Com Deutsche Telekom AG. T-Online Video on Demand. `http://vod.t-online.de/`, 2004-2007.

[29] M. Dick, J. Brandt, V. Kahmann, and L. Wolf. Adaptive Transcoding Proxy Architecture for Video Streaming in Mobile Networks. In *Proceedings of the IEEE International Conference on Image Processing (ICIP 2005)*, volume 3, pages 700–703, Genova, Italy, September 2005.

[30] C. Diot, B. N. Levine, B. Lyles, H. Kassem, and D. Balensiefen. Deployment issues for the IP multicast service and architecture. *IEEE Network*, 14(1):78–88, Jan / Feb 2000.

[31] dom4j - The flexible XML framework for Java. `http://www.dom4j.org`, 2001-2005. BSD Style License by MetaStuff, Ltd.

[32] H.-P. Dommel and J. J. Garcia-Luna-Aceves. Comparison of Floor Control Protocols for Collaborative Multimedia Environments. In *SPIE Symposium on Voice, Video, and Data Communications*, Boston, MA, November 1998.

[33] A. El-Syed, V. Roca, and L. Mathy. A Survey of Proposals for an Alternative Group Communication Service. *IEEE Network*, 17(1):46–51, January 2003.

[34] Benjamin Ernst. Komposition eines Collaborative Streaming Service mit Methoden des Semantic Web. Studienarbeit, Technische Universität Braunschweig, April 2006.

[35] D. Estrin, D. Farinacci, A. Helmy, D. Thaler, S. Deering, M. Handley, V. Jacobson, C. Liu, P. Sharma, and L. Wei. Protocol Independent Multicast-Sparse Mode (PIM-SM): Protocol Specification. RFC 2362, The Internet Engineering Task Force, June 1998.

[36] P. Th. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[37] R. Even and N. Ismail. Conferencing Scenarios. RFC 4597, The Internet Engineering Task Force, July 2006.

[38] D. Ferraiolo and R. Kuhn. Role-Based Access Controls. In *15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.

[39] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, The Internet Engineering Task Force, June 1999.

[40] G. Fortino, W. Russo, C. Mastroianni, C. E. Palau, and M. Esteve. CDN-Supported Collaborative Media Streaming Control. *IEEE Multimedia*, 14(2):60–71, April-June 2007.

[41] UPnP Forum. Device Architecture v 1.0. UPnP Forum Approved Standard, `http://www.upnp.org/resources/documents.asp`, June 2000.

[42] UPnP Forum. MediaServer V 2.0 and MediaRenderer V 2.0. UPnP Forum Approved Standard, `http://www.upnp.org/specs/av/default.asp`, January 2006.

[43] M. Gallant and F. Kossentini. Rate-distortion optimized layered coding with unequal error protection for robust Internet video. *IEEE Transactions on Circuits and Systems for Video Technology*, 11(3):357–372, March 2001.

[44] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[45] W. Geyer and W. Effelsberg. The Digital Lecture Board - A Teaching and Learning Tool for Remote Instruction in Higher Education. In *Proceedings of ED-MEDIA'98*, Freiburg, Germany, 1998.

[46] L. Gharai. RTP with TCP Friendly Rate Control. draft-ietf-avt-tfrc-profile-10, The Internet Engineering Task Force AVT WG, July 2007. Internet Draft, Work in Progress.

[47] C. Griwodz and M. Zink. Dynamic Data Path Reconfiguration. In *International Workshop on Multimedia Middleware 2001 at ACM Multimedia*, pages 72–75, October 2001.

[48] IEEE LTSC P1484.12 LOM Working Group. Draft Standard for Learning Object Metadata. IEEE 1484.12.1-2002, July 2002.

[49] E. Guttman, C. Perkins, and J. Kempf. Service Templates and Service: Schemes. RFC 2609, The Internet Engineering Task Force, June 1999.

[50] E. Guttman, C. Perkins, J. Veizades, and M. Day. Service Location Protocol, Version 2. RFC 2608, The Internet Engineering Task Force, June 1999.

[51] M. Handley, S. Floyd, J. Padhye, and J. Widmer. TCP Friendly Rate Control (TFRC): Protocol Specification. RFC 3448, The Internet Engineering Task Force, January 2003.

[52] M. Handley, V. Jacobson, and C. Perkins. SDP: Session Description Protocol. RFC 4566, The Internet Engineering Task Force, July 2006.

[53] B. Harris, R. Warner, and R. Harris. *The Definitive Guide to SWT and JFace*. APress, 2004.

[54] HAVi Specification 1.0. The HAVi Organization; `http://www.havi.org/`, January 2000.

[55] P. Hoffman and S. Harris. The Tao of IETF: A Novice's Guide to the Internet Engineering Task Force. RFC 4677, FYI 17, The Internet Engineering Task Force, September 2006.

[56] Kabel Deutschland GmbH Homepage. `http://www.kabeldeutschland.com/en.html`.

[57] HomePlug Powerline Alliance, Inc. HomePlug 1.0 Specification, June 2001.

[58] S. Höhne. Signalisierung von Benutzeranforderungen zur Anpassung von Videoströmen. Diploma thesis (in german), Technische Universität Braunschweig, Institut für Betriebssysteme und Rechnerverbund, November 2006.

[59] ICQ, Inc. ICQ Homepage. `http://www.icq.com/`, 1998-2007.

[60] IEEE-SA. IEEE Standard for Information technology - Telecommunications and information exchange between systems - Local and metropolitan area networks - Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications. ISO/IEC 8802-11: 1999, 1999.

[61] IEEE-SA. IEEE Standard for a High Performance Serial Bus (Amendment. IEEE Std 1394a-2000, June 2000.

[62] IEEE-SA. IEEE Standard for a High Performance Serial Bus (High Speed Supplement. IEEE Std 1394b-2002, December 2002.

[63] IEEE-SA. IEEE Standard for Information technology–Telecommunications and information exchange between systems–Local and metropolitan area networks–Specific requirements–Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications. IEEE802.3-2002, 2002.

[64] IEEE-SA. IEEE Standard for Information technology - Telecommunications and information exchange between systems - Local and metropolitan area networks - Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications - Amendment 4. ISO/IEC 8802-11: 1999/Amd 4:2003(E)), 2003.

[65] Ilias Open Source Learning Management System Version 3. EN-202 User Documentation, http://www.ilias.de/, November 2005.

[66] ISO/IEC. Information technology – Generic coding of moving pictures and associated audio information – Part 6: Extensions for DSM-CC. Draft International Standard ISO 13818-6, International Organization for Standardization ISO/IEC JTC1/SC29/WG11, November 1995. Geneva, Switzerland.

[67] ISO/IEC. Information technology - Coding of audio-visual objects - Part 3: Audio. International Standard ISO 14496-3:1999, International Organization for Standardization ISO/IEC JTC1/SC29/WG11, 1999.

[68] ISO/IEC. Coding of audio-visual objects – Systems. International Standard ISO 14496-1, International Organization for Standardization ISO/IEC JTC1/SC29/WG11, 2004. 3rd edition.

[69] ITU-T. Overview of Digital Subscriber Line (DSL) Recommendations. ITU-T Recommendation G.995.1, February 2001. Series G: Transmission Systems and Media, Digital Systems and Networks.

[70] ITU-T. Phoneline networking transceivers – Enhanced physical, medium access and link layer specifications. ITU-T Recommendation G.9954, February 2005. Series G: Transmission Systems and Media, Digital Systems and Networks.

[71] ITU Telecommunication Standardization Sector (ITU-T). Narrow-band visual telephone systems and terminal equipment. ITU-T Recommendation H.320, March 2004.

[72] ITU Telecommunication Standardization Sector (ITU-T). Advanced video coding for generic audiovisual services. ITU-T Recommendation H.264, March 2005. also published as MPEG-4, Part 10: ISO/IEC 14496-10:2005.

[73] ITU Telecommunication Standardization Sector (ITU-T). IPTV Focus Group (FG IPTV). http://www.itu.int/ITU-T/IPTV/, 2007.

[74] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose, and D. Towsley. Inferring TCP connection characteristics through passive measurements. In *IEEE INFOCOM 2004, The 23rd Annual Joint Conference of the IEEE Computer and Communications Societies*, march 2004.

[75] T. Jehaes, D. De Vleeschauwer, T. Coppens, B. van Doorselaer, E. Deckers, W. Naudts, K. Spruyt, and R. Smets. Access network delay in networked games. In *NetGames '03: Proceedings of the 2nd workshop on Network and system support for games*, pages 63–71, New York, NY, USA, 2003. ACM Press.

[76] W. J. Jeon and K. Nahrstedt. QoS-aware Middleware Support for Collaborative Multimedia Streaming and Caching Service. *Microprocessors and Microsystems, Special Issue on QoS-enabled Multimedia Provisioning over the Internet, Elsevier Science*, December 2002.

[77] A. Johnston and O. Levin. Session Initiation Protocol (SIP) Call Control – Conferencing for User Agents. RFC 4579, The Internet Engineering Task Force, August 2006.

[78] V. Kahmann, J. Brandt, and L. Wolf. Flexible Media Reflection for Collaborative Streaming Scenarios. In *WCW '05: Proceedings of the 10th International Workshop on Web Content Caching and Distribution (WCW'05)*, pages 71–76, Washington, DC, USA, 2005. IEEE Computer Society.

[79] V. Kahmann and L. Wolf. Collaborative Media Streaming in an In-Home Network. In *Proc. of International Workshop on Smart Appliances and Wearable Computing (IWSAWC)*, Phoenix/Mesa, Arizona, April 2001.

[80] W. Kellerer, M. Wagner, W.-T. Balke, and H. Schulzrinne. Preference-Based Session Management for IP-based Mobile Multimedia Signaling. *European Transactions on Telecommunications*, 2004.

[81] J. Kempf and J. Rosenberg. SIP Abstract Service Type. IANA Assignment, http://www.iana.org/assignments/svrloc-templates/sip.1.0.en, 1997.

[82] J. Kempf and J. Rosenberg. SIP Proxy/Redirect Concrete Service Type. IANA Assignment, http://www.iana.org/assignments/svrloc-templates/sipproxy.1.0.en, 1997.

[83] J. Kempf and J. Rosenberg. SIP Registrar Concrete Service Type. IANA Assignment, http://www.iana.org/assignments/svrloc-templates/sipregistrar.1.0.en, 1997.

[84] Y. Kikuchi, T. Nomuar, S. Fukunage, Y. Matsui, and H. Kimata. RTP Payload Format for MPEG-4 Audio/Visual Streams. RFC 3016, The Internet Engineering Task Force, November 2000.

[85] P. T. Kirstein and R. Bennett. Multimedia Education and Conferencing Collaboration over ATM Networks and Others (MECCANO). Final Report, RE 4007, June 2000.

[86] C. D. Knutson and J. M. Brown. *IrDA Principles and Protocols: The IrDA Library, Volume 1*. MCL Press, 2004.

[87] G. Kortuem, C. Kray, and H. Gellersen. Sensing and visualizing spatial relations of mobile devices. In *UIST '05: Proceedings of the 18th annual ACM symposium on User interface software and technology*, pages 93–102, New York, NY, USA, 2005. ACM Press.

[88] T. Kraft. Entwicklung eines Gateways zur Nutzung mehrerer Service-Discovery-Protokolle. Diploma thesis (in german), Technische Universität Braunschweig, Institut für Betriebssysteme und Rechnerverbund / Universität Karlsruhe, AIFB, July 2002.

[89] D. Kruglyakov. Auffinden von Diensten und Nutzern in einer Collaborative-Streaming-Umgebung. Diploma thesis (in german), Technische Universität Braunschweig, Institut für Betriebssysteme und Rechnerverbund, May 2005.

[90] D. R. J. Laming. *Information Theory of Choice-Reaction Times*. Academic Press, London, 1968.

[91] J. Leigh, T. A. DeFanti, A. E. Johnson, M. D. Brown, and D. J. Sandin. Global Tele-Immersion: Better Than Being There. In *7th International Conference on Artificial Reality and Tele-Existence (ICAT)*, pages 10–17, Tokyo, Japan, December 1997.

[92] Skype Limited. Skype Official Website. `http://www.skype.com`, 2007 (last change).

[93] LIVE555 Streaming Media library. `http://www.live.com/liveMedia/`, October 2005.

[94] M. Lohse, M. Repplinger, and P. Slusallek. An Open Middleware Architecture for Network-Integrated Multimedia. In *Proceedings of the Joint International Workshops on Interactive Distributed Multimedia Systems and Protocols for Multimedia Systems (IDMS/PROMS) 2002*, volume 2515 of *Lecture Notes in Computer Science*, pages 327–338. Springer, 2002.

[95] M. Lohse, M. Repplinger, and P. Slusallek. Dynamic Distributed Multimedia: Seamless Sharing and Reconfiguration of Multimedia Flow Graphs. In *Proceedings of the 2nd International Conference on Mobile and Ubiquitous Multimedia (MUM 2003)*, pages 89–95. ACM Press, 2003.

[96] L. Martensen. Multimedia Caching. Diploma thesis (in german), Technische Universität Braunschweig, Institut für Betriebssysteme und Rechnerverbund, September 2006.

[97] D. Martin, M. Burstein, J. Hobbs, O. Lassila, Drew McDermott, Sheila McIlraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry Payne, Evren Sirin, Naveen Srinivasan, and Katia Sycara. OWL-S: Semantic Markup for Web Services. Technical Overview White Paper, `http://www.daml.org/services/owl-s/1.1/overview`, November 2004. OWL-S Release 1.1.

[98] L. Mathy, G. Leduc, O. Bonaventure, and A. Danthine. A Group Communication Framework. In W. Bauerfeld, O. Spaniol, and F. Williams, editors, *Broadband Islands '94: Connecting with the End-User, Proceedings of the 3rd Int'l Conference*, pages 167–178, Hamburg, Germany, June 1994.

[99] S. McCanne, M. Vetterli, and V. Jacobson. Low-complexity video coding for receiver-driven layered multicast. *IEEE Journal on Selected Areas in Communications*, 15(6):983–1001, August 1997.

[100] A. Meissner, S. B. Musunoori, and L. Wolf. MGMS/GML – Towards a new Policy Specification Framework for Multicast Group Integrity. In *Proceedings of 2004 Symposium on Applications and the Internet (SAINT 2004)*, pages 233 – 239, Tokyo, Japan, January 26-30 2004.

[101] A. Meissner, L. Wolf, and R. Steinmetz. A novel Group Integrity Concept for Multimedia Multicasting. In *IDMS 2001*, pages 233 – 244, Lancaster, UK, September 4-7 2001.

[102] Digital Video Broadcasting (DVB); Multimedia Home Platform (MHP) Specification 1.1.1. ETSI TS 101 812 V1.2.1, `http://www.etsi.org`, June 2003. RTS/JTC-DVB-149.

[103] Microsoft. Microsoft Site Server. `http://www.microsoft.com/windows/netmeeting/ils/`, May 2001.

[104] D. Mills. Network Time Protocol (Version 3) Specification, Implementation and Analysis. RFC 1305, The Internet Engineering Task Force, March 1992.

[105] A. Moizard. The GNU oSIP library. `http://www.gnu.org/software/osip`, 2002.

[106] M. Ringel Morris, K. Ryall, C. Shen, C. Forlines, and F. Vernier. Beyond "Social Protocols": Multi-User Coordination Policies for Co-located Groupware. In *CSCW '04*, Chicago, Illinois, USA, November 2004. ACM.

[107] Multimedia Framework (MPEG-21). ISO/IEC JTC 1.29.17 (21000), 2004.

[108] Multimedia Content Description Interface (MPEG-7). ISO/IEC JTC 1.29.15 (15938), 2002.

[109] MPEG4IP Open Streaming Video and Audio, release 1.3. http://mpeg4ip.sourceforge.net/, May 2005.

[110] S. Mungee, N. Surendran, and D. C. Schmidt. The Design and Performnce of a CORBA Audio/Video Streaming Service. In *Proceedings of HICSS-32 International Conference on System Sciences, minitrack on Multimedia DBMS and the WWW*, Hawaii, January 1999.

[111] K. Nahrstedt and W. Balke. A Framework for Service Composition. In *ACM Multimedia 2004*, pages 181 – 185, Portland, October 2004. ACM.

[112] K. Nakane, Y. Sato, Y. Kiyose, M. Shimamoto, and M. Ogawa. Development of combined HDD and recordable-DVD video recorder. In *International Conference on Consumer Electronics (ICCE) 2002*, pages 264–265, 2002.

[113] A. Niemi. Session Initiation Protocol (SIP) Extension for Event State Publication. RFC 3903, The Internet Engineering Task Force, October 2004.

[114] O. Novo, G. Camarillo, D. Morgan, and R. Even. Conference Information Data Model for Centralized Conferencing (XCON). draft-ietf-xcon-common-data-model-11.txt, The Internet Engineering Task Force XCON WG, June 2008. Internet Draft, Work in Progress.

[115] Department of Defense. Trusted Computer Security Evaluation Criteria. DOD 5200.28-STD, 1985.

[116] Joanneum Research Institute of Information Systems & Information Management. MPEG-7 Resources. http://mpeg-7.joanneum.at/, 2004-2007. Library Version 2.1 implements ISO/IEC 15938:2001.

[117] Society of Motion Picture and Television Engineers. Television, Audio and Film - Time and Control Code. SMPTE 12M-1999.

[118] National Institute of Standards and Technology (NIST). Project IP Telephony / VoIP. http://snad.ncsl.nist.gov/proj/iptel/, 2002 – 2006.

[119] OpenSLP project page. http://www.openslp.org, 2000-2005. BSD Style License by Caldera Systems, Inc.

[120] K. S. Park, A. Kapoor, C. Scharver, and J. Leigh. Exploiting Multiple Perspective in Tele-Immersion. In *Proceedings of IPT 2000*, Ames, Iowa, June 2000.

[121] C. Peltz. Web services orchestration and choreography. *IEEE Computer*, 36(10):46–52, October 2003.

[122] GNU Telephony Project. GNU ccRTP. http://www.gnu.org/software/ccrtp/, November 2007. Version 1.6.0.

[123] The MPlayer Project. MPlayer – The Movie Player. http://www.mplayerhq.hu, October 2006. Version 1.0rc1.

[124] 3GPP TSG/WG R2. High Speed Downlink Packet Access (HSDPA); Overall description; Stage 2. 3rd Generation Partnership Project, Technical Specification 25.308, 2002.

[125] RealNetworks, Inc. Helix Player 1.0.8 Gold. https://player.helixcommunity.org/, August 2006.

[126] RealNetworks, Inc. Media Players. http://www.realnetworks.com/products/media_players.html, 2007.

[127] RealNetworks, Inc. RealNetworks (R) Media Servers. http://www.realnetworks.com/products/media_delivery.html, 2007.

[128] T. Reenskaug. Models – Views – Controllers. Technical report, Xerox PARC, December 1979.

[129] reSIProcate.org. ReSIProcate project page. http://www.resiprocate.org, May 2007.

[130] A. B. Roach. Session Initiation Protocol (SIP)-Specific Event Notification. RFC 3265, The Internet Engineering Task Force, June 2002.

[131] J. Rosenberg. A Presence Event Package for the Session Initiation Protocol (SIP). RFC 3856, The Internet Engineering Task Force, August 2004.

[132] J. Rosenberg. A Framework for Conferencing with the Session Initiation Protocol. RFC 4353, The Internet Engineering Task Force, February 2006.

[133] J. Rosenberg and H. Schulzrinne. An Offer/Answer Model with the Session Description Protocol (SDP). RFC 3264, The Internet Engineering Task Force, June 2002.

[134] J. Rosenberg and H. Schulzrinne. Session Initiation Protocol (SIP): Locating SIP Servers. RFC 3263, The Internet Engineering Task Force, June 2002.

[135] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. RFC 3261, The Internet Engineering Task Force, June 2002.

[136] J. Rosenberg, H. Schulzrinne, and P. Kyzivat. Caller Preferences for the Session Initiation Protocol (SIP). RFC 3841, The Internet Engineering Task Force, August 2004.

[137] J. Rosenberg, H. Schulzrinne, and O. Levin. A Session Initiation Protocol (SIP) Event Package for Conference State. RFC 4575, The Internet Engineering Task Force, August 2006.

[138] J. Rosenberg, H. Schulzrinne, and R. Mahy. An INVITE-Initiated Dialog Event Package for the Session Initiation Protocol (SIP). RFC 4235, The Internet Engineering Task Force, November 2005.

[139] K. Rothermel and T. Helbig. An Adaptive Stream Synchronization Protocol . In *Proc. of 5th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, Durham, New Hampshire, USA, April 1995.

[140] R. S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman. Role-Based Access Control Models. *IEEE Computer*, 29(2):38–47, 1996.

[141] D. Sandras. Ekiga Project Page. `http://www.ekiga.org`, April 2007. Software Release Version 2.0.9.

[142] Antisip SARL. The eXtended osip library. `http://savannah.nongnu.org/projects/exosip`, April 2006.

[143] N. Scheele, M. Mauve, W. Effelsberg, A. Wessels, H. Horz, and S. Fries. The Interactive Lecture: A New Teaching Paradigm Based on Ubiquitous Computing. In *CSCL 2003 (Computer Support for Collaborative Learning)*, June 2003. Poster Presentation.

[144] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 3550, The Internet Engineering Task Force, July 2003.

[145] H. Schulzrinne, A. Rao, and R. Lanphier. Real Time Streaming Protocol (RTSP). RFC 2326, The Internet Engineering Task Force, April 1998.

[146] H. Schulzrinne, A. Rao, R. Lanphier, M. Westerlund, A. Narasimhan, and M. Stiemerling. Real Time Streaming Protocol 2.0 (RTSP). draft-ietf-mmusic-rfc2326bis-15.txt, IETF MMUSIC WG, June 2007.

[147] F. B. Segui, J. C. G. Cebollada, and J. L. Mauri. Multimedia Group Synchronization Algorithm Based on RTP/RTCP. In *Proceedings of the Eighth IEEE International Symposium on Multimedia (ISM'06)*, pages 754–757, December 2006.

[148] Bluetooth SIG. Bluetooth Core Specification. `http://www.bluetooth.com/Bluetooth/Learn/Technology/Specifications/`, November 2004. Version 2.0 + EDR.

[149] K. Singh and H. Schulzrinne. Peer-to-peer Internet Telephony using SIP. In *NOSSDAV '05: Proceedings of the international workshop on Network and operating systems support for digital audio and video*, pages 63–68, New York, NY, USA, June 2005. ACM Press.

[150] C. Southeren and D. Sandras. OPAL - Open Phone Abstraction Library. `http://www.voxgratia.org/`, January 2007. Stable Phobos Release 4, version 2.2.4.

[151] R. Sparks. The Session Initiation Protocol (SIP) Refer Method. RFC 3515, The Internet Engineering Task Force, April 2003.

[152] R. Steinmetz. Human perception of jitter and media synchronization. *IEEE Journal of Selected Areas in Communication (JSAC)*, 14(1):61–72, 1996.

[153] I. Stoica, T. S. Eugene Ng, and H. Zhang. REUNITE: A Recursive Unicast Approach to Multicast. In *Proceedings IEEE INFOCOM 2000, The Conference on Computer Communications, Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 1644–1653, Tel Aviv, Israel, March 2000.

[154] Sun. Java Media Framework API (JMF), version 2.1.1e. `http://java.sun.com/products/java-media/jmf/index.jsp`, May 2003.

[155] Sun Microsystems, Inc. JAIN and Java in Communications. JAIN Reference / White Papers, `http://java.sun.com/products/jain`, March 2004.

[156] Sun Microsystems, Inc. Jini Specifications, v.2.1. `http://java.sun.com/products/jini/`, 2005.

[157] ADL Technical Team. Sharable Content Object Reference Model (SCORM) 2004 2nd Edition Document Suite. Advanced Distributed Learning, `http://www.adlnet.org`, July 2004.

[158] The Object Management Group, Inc. Audio/Visual Stream Specification V1.0. OMG Document formal/00-01-03, June 1998.

[159] International Telecommunication Union. Packet-based multimedia communications systems - ITU-T Recommendation H.323. Telecommunication Standardization Sector of ITU, February 1998.

[160] 3GPP TS 24.228 V5.13.0. Signalling flows for the IP multimedia call control based on SIP and SDP; Stage 3 (Release 5). 3rd Generation Partnership Project, Technical Specification Group Core Network and Terminals, June 2005. IMS Technical Specification.

[161] K. Velitchkov. Entwurf und Implementierung eines flexiblen, modularen Association Service. Diploma thesis, Technische Universität Braunschweig, Institut für Betriebssysteme und Rechnerverbund, May 2005.

[162] A. Vetro, C. Christopoulos, and H. Sun. Video transcoding architectures and techniques: an overview. *Signal Processing Magazine, IEEE*, 20(2):18–29, March 2003.

[163] VideoLAN. VLC Media Player. `http://www.videolan.org/`, April 2007. Version 0.8.6b.

[164] J. Vogel, M. Mauve, W. Geyer, V. Hilt, and C. Kuhmünch. A Generic Late Join Service for Distributed Interactive Media. In *Proc of the 8th ACM Multimedia, ACM MM 2000*, Los Angeles, USA, 2000.

[165] Vovida.org. VOCAL 1.5.0. `http://www.vovida.org/applications/downloads/vocal/`, April 2003.

[166] W3C. Web Services Architecture. W3C Working Group Note, `http://www.w3.org/TR/ws-arch/`, February 2004.

[167] W3C. Synchronized Multimedia Integration Language (SMIL 2.1). W3C Recommendation 13 December 2005, `http://www.w3.org/TR/SMIL2/`, December 2005.

[168] W3C. SOAP Version 1.2. W3C Recommendation, `http://www.w3.org/TR/soap/`, April 2007.

[169] M. Wahl, T. Howes, and S. Kille. Lightweight Directory Access Protocol (v3). RFC 2251, The Internet Engineering Task Force, December 1997.

[170] J. S. P. Wandelmer. Omnividea FOBS - FFMpeg C++ and JMF Bindings, version 0.4.1. `http://fobs.sourceforge.net/index.html`, January 2007.

[171] Y. Xie, C. Liu, M. J. Lee, and T. N. Saadawi. Adaptive multimedia synchronization in a teleconference system. *Multimedia Systems*, 7(4):326–337, July 1999. Springer Berlin / Heidelberg.

[172] D. Yang, A. El Saddik, and N. D. Georganas. "Latecomer Support and Client Synchronization for Synchronous Multimedia Collaborative Environments". In *Proceedings of the 4th International Workshop on Collaborative Editing*, New Orleans, USA, Nov 2002. Adjunct to the ACM Conference on Computer Supported Cooperative Work.

[173] W. Zhao and H. Schulzrinne. Enhancing Service Location Protocol for Efficiency, Scalability and Advanced Discovery. *Journal of Systems and Software*, 75(1-2):193–204, February 2005.

[174] W. Zhao, H. Schulzrinne, and E. Guttman. Mesh-enhanced Service Location Protocol (mSLP). RFC 3528, April 2003.

[175] W. Zhao, H. Schulzrinne, E. Guttman, C. Bisdikian, and W. Jerome. Remote Service Discovery in the Service Location Protocol (SLP) via DNS SRV. RFC 3832, The Internet Engineering Task Force, July 2004.